University of Zurich[UZH]

Department of Informatics

Supporting Requirements Update During Software Evolution

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of Doctor of Science

by
Eya Ben Charrada
from Tunisia

Prof. Dr. Martin Glinz
Prof. Dr. Gustavo Alonso

2013

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, April 3, 2013

Head of the Ph.D. committee for informatics: Prof. Abraham Bernstein, Ph.D.

Acknowledgement

He who does not thank people does not thank God.

Prophet Muhammad

After having invested four years of my life in preparing this work, I cannot but agree with the already spread belief that completing a PhD is a long and demanding journey, both intellectually and emotionally. What I also learned is that, although very important, commitment and perseverance are not enough to achieve such demanding goals. In fact, what is also essential, are the people in the way that support you to arrive to the end. I am grateful to all the people that supported me in my journey.

I want to thank Professor Martin Glinz for giving me the opportunity to do my thesis in his research group. The work environment was excellent and I enjoyed the freedom he gave us to explore new ideas. The feedback he provided on my papers was always

valuable and covered both the general aspects and the very details of them.

I want to thank Professor Gustavo Alonso, who guided me through my very first steps into research during my Master Thesis, which I did in his group. I am also thankful to him for accepting to co-advise this work.

I want to thank my colleagues at the RERG group for the enriching and inspiring discussions. Many great ideas came out of the various interesting discussions I had with Cédric Jeanneret about my topic and about research in general. I also learned different writing skills during my collaboration with Anne Koziolek. I want to thank Samuel Fricker, Tobias Reinhard, Arun Mukhija, Cédric Jeanneret and Dustin Wüest for reviewing my papers and providing valuable feedback. I am also thankful to Norbert Seyff, Reinhard Stoiber, Irina Todoran and Nicolas Hoby for the comments they gave me on my rehearsals and/or my research.

Some ideas in this research were implemented by students from the University of Zurich. I would especially like to thank Simon Käser for implementing the differencing tool that we used in the experiments and David Caspar for preparing the data and the traceability links that allowed building the traceability benchmark.

Last but not least, I want to thank my parents, my husband and his family, my brothers and my friends for continuously supporting me to move forward and achieve my goals.

Abstract

The existence of an up-to-date requirements specification is very crucial for software evolution. In fact, the requirements specification facilitates program comprehension, gives the rationale behind the implementation, prevents undoing important decisions and serves as a basis to discuss feature changes with stakeholders. Despite their importance, requirements specifications are rarely kept up-to-date. This is mainly due to the high-costs of the approaches used nowadays to maintain them. In fact, requirements maintenance is still a manual task that requires the engineer to go through the whole requirements document, which can be hundreds or thousands of pages, and look for the parts that need to be changed. In this thesis, we explore new ways to reduce the effort needed to maintain the requirements specification. We propose two approaches, one based on tests and one based on code, to automatically identify the parts of the requirements specification that become outdated when the implementation is changed. In the code-based approach, which is the main contribution of the thesis, we compare two versions of the source code, identify the relevant

changes and trace these changes back to the requirements to identify the parts that are impacted. When applying the approach to two case studies, it identified 70% to 100% of the outdated requirements within a list including less than 20% of the total number of requirements in the specification. Automatically identifying the requirements that are likely to be outdated should reduce the effort needed for requirements maintenance and thus encourage maintainers to regularly update the specification.

Zusammenfassung

Die Verfügbarkeit einer aktuellen Anforderungsspezifikation ist von grosser Bedeutung für die Software-Evolution. Eine Anforderungsspezifikation erleichtert das Programmverständnis, liefert die Beweggründe für die Implementierung, verhindert, dass wichtige Entscheidungen versehentlich rückgängig gemacht werden und dient als Grundlage zur Diskussion von Funktionsänderungen mit Interesseneignern (Stakeholdern). In realen Projekten allerdings werden Anforderungsspezifikationen selten gepflegt bzw. aktuell gehalten. Dies ist vor allem bedingt durch den hohen Aufwand, den die heutigen Ansätze erfordern. Faktisch ist die Pflege der Anforderungsspezifikation heute noch ein manueller Prozess, bei dem die verantwortliche Person das gesamte Anforderungsdokument, welches hunderte oder tausende von Seiten umfassen kann, durchlesen muss, um die zu aktualisierenden Stellen zu identifizieren. In dieser Arbeit untersuchen wir neue Ansätze, um den Aufwand für die Pflege der Anforderungsspezifikation zu reduzieren. Wir schlagen zwei automatisierte Methoden vor, um diejenigen Teile der Anforderungsspezifikation zu identifizieren, die bei einer

Änderung in der Implementierung angepasst werden müssen. Eine der beiden Methoden ist testbasiert, die andere quellcodebasiert. In der quellcodebasierten Methode, die den wichtigsten Beitrag dieser Arbeit darstellt, werden zwei Versionen des Quellcodes verglichen, relevante Änderungen identifiziert und diese zu den zugehörigen Anforderungen zurückverfolgt. Bei der Anwendung dieser Methode auf zwei Fallstudien wurden 70% bis 100% der nicht mehr aktuellen Anforderungen identifiziert, und dies in einer Liste mit weniger als 20% der gesamten Anforderungen. Die automatische Ermittlung der Anforderungen, welche bei einer Änderung im Code mit grosser Wahrscheinlichkeit nicht mehr aktuell sind, sollte den Aufwand für die Pflege der Anforderungen reduzieren und dadurch die Pflegeverantwortlichen motivieren, nicht nur den Code, sondern auch die Anforderungsspezifikation zu pflegen und aktuell zu halten.

Contents

List of Tables

List of Figures

Chapter 1
Synopsis

1.1 Problem Statement and Goal

Keeping the requirements specification up-to-date when software systems evolve is crucial for software maintainability. In fact, the requirements specification gives a high-level view of the system that is much easier to understand than code. Requirements specifications also support program comprehension by providing the rationale behind the implementation. Understanding a system from its parts only can be very challenging and requires complex reasoning, as the intent is missing [Lev00]. However, when the rationale is known, comprehending the system becomes straightforward. Having the rationale behind the implementation is also very important as it prevents undoing important decisions. Requirements, which are usually written in natural language, can also be used as a basis to discuss changes, as they are understood

by all stakeholders including those that are not from the software engineering field.

Updating the requirements specification is still a manual task which is expensive, time-consuming and error-prone. Ideally, the maintainer should first do an impact analysis at the requirements level and then implement these changes in the code. There are, however, two problems faced by the maintainer when performing such a task. First, if the requirements specification is long (e.g. hundreds or thousands of pages), then manually identifying the impacted requirements is likely to require considerable time and effort. Second, when implementing the changes, the maintainer needs to do a second impact analysis at the code level. This is because the source code includes several architectural and implementation details that are not addressed at the requirements level and thus changes in these elements will be missed when tracing the changes in the requirements to the source code.

To avoid the extensive effort of doing the impact analysis twice, maintainers usually apply the changes to the source code directly and leave the requirements unchanged. This results in the requirements specification becoming obsolete and useless. Losing the knowledge contained in the requirements specification hinders the maintainability of the system and leads it to enter a stage where only minor changes can be applied to it [BR00].

Goal of the thesis

The goal of this thesis is to identify new ways to reduce the effort required to maintain the requirements specification and thus

encourage maintainers to keep the specification up-to-date when a software system evolves. We focus on the maintenance of *functional* requirements.

1.2 State of the Art

Keeping the requirements specification up-to-date when software systems evolve is recognized as a challenge by researchers and experts in the fields of requirements engineering and software evolution. In a survey about the use of software documentation, Lethbridge et al. [LSF03] report that requirements are rarely updated in practice. When the requirements are updated, the update is usually done weeks after the code is changed. The problem of obsolete requirements has also been mentioned, in a less formal way, in many publications in the fields of software engineering, requirements engineering and software evolution (e.g. [BR00] [You05] [MWD+05]).

In the remainder of this section, we discuss relevant literature about managing the evolution of requirements, co-evolution, reverse engineering requirements from the implementation, impact analysis, change propagation and traceability.

1.2.1 Managing Requirements Evolution

Literature about requirements evolution covers several aspects such as specifying new requirements, keeping the requirements specification consistent, reusing existing solutions, etc.

"How to specify new requirements for existing systems?" is the question addressed by the work of Herrmann et al. in [HWP09]. They explore a way to specify delta requirements to support software evolution. Their approach is meant to address the problem of specifying requirements changes when a reliable requirements specification is missing. In their approach they extend TORE [PK04] and use it to specify delta requirements in detail while describing the rest of the system on a higher-level of granularity.

In [ZG03], Zowghi and Gervasi address the question of how to ensure that the requirements specification is correct, consistent and complete after each change. They present different validation checks that can be used to detect errors and problems in the requirements specification.

In [EBJ11], Ernst et al. explore how to reuse existing solutions to address changes in requirements. Re-implementing a new optimal solution for a problem whenever an unanticipated change is requested can be very expensive. Therefore, the authors explore the possibility of modifying and reusing as much as possible of the existing solutions and only build incremental repairs to address changes in requirements.

Most of the existing works about managing requirements evolution focus on the requirements part only. What is still missing, however, are ways to *support the propagation of the changes between the requirements and the code*. Ignoring the propagation part means that the maintainer has to do the task manually. Therefore, these approaches do not solve the problem we mentioned in Section 1.1 about doing the impact analysis twice.

1.2.2 Co-evolution

As our goal is to ensure the co-evolution of the code and the requirements specification, we explored the existing work that addresses the co-evolution of software artifacts. Most of the approaches we found address the co-evolution of the implementation with the design [MKPW06] [CPGS07] [DVMW02]. Managing the co-evolution of the requirements specification and the implementation is very different from the co-evolution of design and implementation because the requirements specification is written in natural language. This makes the analysis of the changes and their propagation to the code more challenging than when using formal design and architectural documents. Additionally, design documents are usually similar to the implementation as both relate to the solution domain. The requirements, however, are different because they relate to the problem domain and thus the mapping between requirements and code becomes more complex.

In [ES05], Etien and Salinesi address the problem of keeping the alignment of the requirements with the other software entities. Their approach for propagating the changes is based on the use of traceability links that connect related entities. We discuss the use of traceability linking to propagate changes between artifacts in more detail in Section 1.2.4.

1.2.3 Reverse Engineering

One way to get requirements that are up-to-date with the implementation is to reverse engineer the requirements specification from

the code. In [YWM+05], Yu et al. propose an approach to reverse engineer goal models from source code. Di Lucca et al. [LFdC00] propose an approach for recovering use-case models from source code. In [ERSS02], El-Ramly et al. propose to use system-user interaction traces to recover software requirements.

The main limitation of reverse engineering approaches is that the generated artifacts are either imprecise or incomplete [CDP07]. Regenerating the requirements specification from code can be very interesting if no requirements specification exists. However, if the specification exists, then finding a way to keep it up-to-date is likely to result in a specification that is of much higher quality than a reverse engineered one.

1.2.4 Impact Analysis and Change Propagation

Our work relates to impact analysis and change propagation, as it is meant to support the maintainer in identifying impacted parts and in propagating the changes between them. Most impact analysis approaches focus on identifying the impact of a change either at the implementation level [LR03] [LOA00] [Boh02] [AOH05] [HH04] or at the requirements level [SHR07] [HRH05] [LDLL10], but do not address the propagation of changes between requirements and code.

In [SW08], Sherriff and Williams propose to use the history of changed files to group related documents, and thus facilitate the change propagation later. The idea behind the approach is the

following: if file A and file B tend to change together, then files A and B are considered as related. This approach can be used to link the code and the requirements. As the approach requires a history of changes that were applied to both requirements and code, it cannot be used when only the code is maintained. However, this approach is likely to be very useful when used in combination with other approaches for supporting the co-evolution of requirements and code.

The main approach we found for propagating changes vertically between requirements and the code is traceability. We dedicate the rest of this section to present traceability approaches.

Traceability Software traceability is defined as "*the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artefacts, and the rationale that explains the form of the artefacts*"[SZ05].

Software traceability has been a very active field of research in the last decades. Previous work in traceability includes an analysis of the requirements traceability problem [GF94], exploration of reference models for requirements traceability [RJ01], automatic generation of traceability links [ACCDL00], management of traces evolution [CHCC03], automatic maintenance of traces [MG12] and tools for managing and generating traceability links [HDS+07].

Although very important for managing the evolution of software systems, the use of traceability is still very limited in practice. This is because defining and maintaining traceability links is a very expensive and time-consuming task. To address this problem, various approaches for supporting the generation and maintenance of traceability links have been developed.

Many automated tracing approaches are based on information retrieval: the similarity between artifacts is computed and used to generate candidate traceability links [ACC+02] [HDS06] [MM03] [LFOT07]. To improve the obtained results, information retrieval methods have also been combined with machine learning [CHCGE10], execution tracing [EAAG08], and analyst feedback [HDS05]. One of the main limitations of these approaches is that the obtained precision is usually low (many false positives are included) when the recall is high. Therefore, the developer will have to re-check and correct the links manually after the generation. In [Egy03], Egyed proposes an approach that uses trace analysis to generate and validate traceability links between artifacts. The approach is semi-automated, as it requires an initial set of traces that are manually defined between the elements to be traced. Additionally, a set of scenarios that will be executed against the code is required.

When reliable traceability links exist, they help reduce the effort required to propagate changes from one artifact to another. However, there are two main limitations for traceability when used to propagate changes between requirements and code. First, traceability links are usually not available due to the costs needed to implement them. Second, there is usually much scattering and tangling

between requirements and code. This results in a high number of links that can be overwhelming for the maintainer.

1.2.5 State of the Art and Thesis Contribution

The approaches presented earlier for managing the evolution of requirements, co-evolution and reverse engineering, do not address the problem of propagating changes between the requirements and the code when software system evolve. Therefore, these approaches do not solve the problem we are addressing.

Software traceability is very useful for supporting change propagation between artifacts as it facilitates the navigation between the elements that are related to each other. However, traceability has two main limitations. First, defining traceability links that are complete and correct is time-consuming and expensive. Therefore, traceability links are not available in most real-world projects. Second, the use of traceability links between requirements and code can be challenging, as there is much scattering and tangling between these two artifacts. The scattering and tangling is likely to result in a very high number of links that can be overwhelming and confusing for the maintainer.

In this thesis, we aim at going beyond simple traceability for reducing the effort needed when propagating changes between requirements and source code as presented in Figure 1.1. Our work does not fully automate the propagation of change. Instead, we support change propagation by automatically identifying the requirements that are likely to be impacted when the code changes.

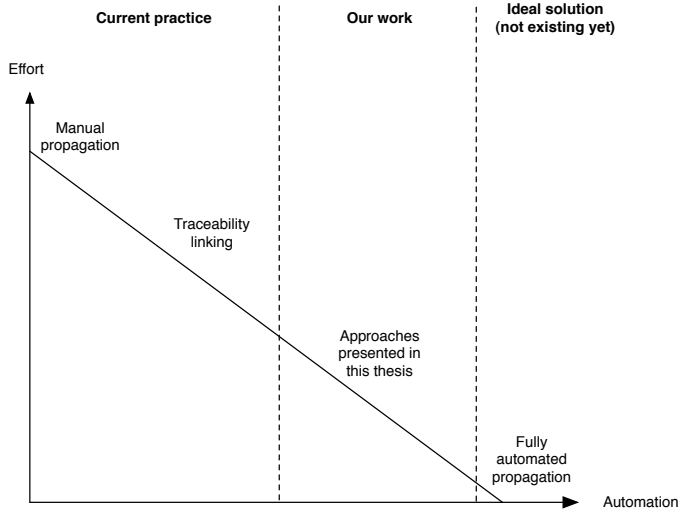

Figure 1.1: Propagating changes between requirements and code: relation between current practice and the thesis

1.3 Research Questions

The goal of the thesis is to support the maintainer in the task of requirements update. We mainly focus on the evolution of functional requirements. The general question we aim to answer in order to achieve the thesis goal is the following:

> What is an effective way to obtain automatic hints about changes in requirements that will guide the maintainer during the requirements update task?

To answer this question, we explored the following two directions:

- How can we support requirements maintenance using the changes in source code?
- How can we support requirements maintenance using the changes in high-order tests?

We defined five research questions, where three of them (RQ 1, RQ 2 and RQ 3) relate to the use of changes in source code to automatically identify changes in requirements and two of them (RQ 4 and RQ 5) address the use of acceptance tests as an intermediate between requirements and code to support requirements change. An overview of the research questions and their relations to each other is presented in Figure 1.2.

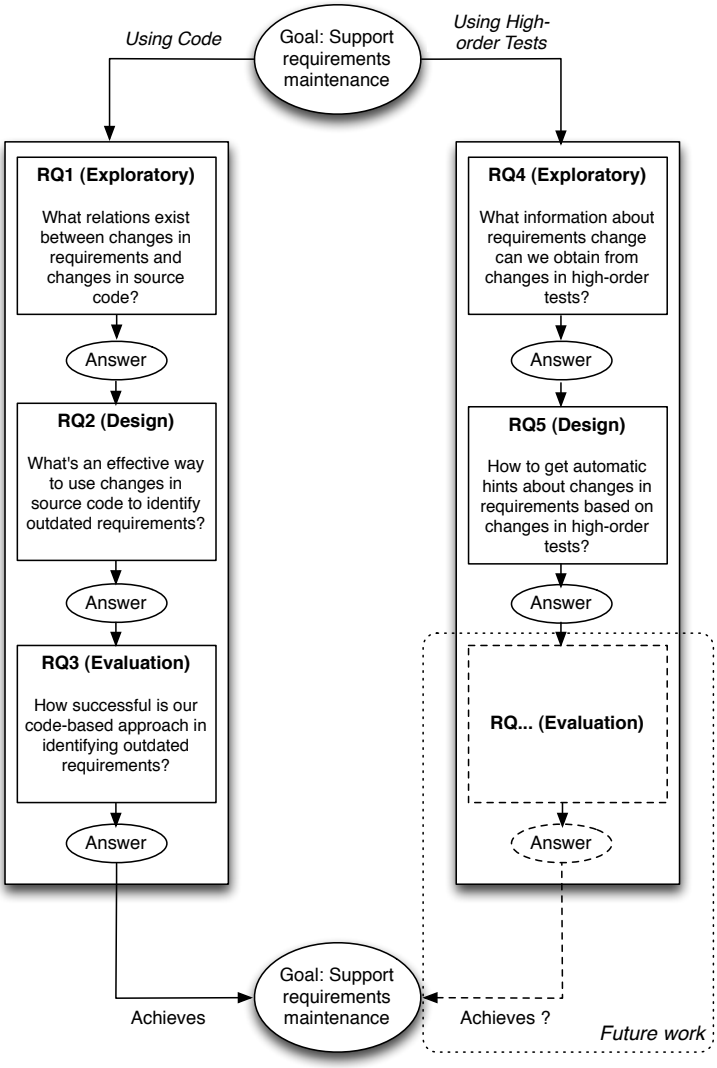

Figure 1.2: Research questions

1.3.1 Research Direction I: Using Changes in Code to Identify Outdated Requirements

RQ 1: What relations exist between changes in requirements and changes in source code? (Exploratory question) This research question aims at exploring the relations that exist between changes in source code and changes in requirements and identifying the code changes that are likely to impact requirements. There are different types of changes that are usually applied to the source code. Some code changes impact the requirements and some do not, such as bug fixes, refactorings, or changes in architectural and implementation details. As we would like to automatically identify whether or not a requirements specification needs to be changed when the implementation changes, we need to find a way to differentiate between the code changes that impact requirements and the changes that do not.

To address this question, we conducted an exploratory study on an open source software project, where we looked for patterns of the code changes that impact the external behaviour of the system. In the study, we assume that changes in the external behaviour of a software system relate to changes in its functional requirements.

RQ 2: What is an effective way to use changes in source code to identify outdated requirements? (Design question) This question depends on RQ 1. The answer to RQ 1 is a set of observations about the relations between changes in the

code and changes in requirements. In RQ 2, we aim at using the findings from RQ 1 to build an effective approach for identifying outdated requirements automatically by analysing source code changes.

We address RQ 2 in two steps. In the first step, we use the answers from RQ 1 to build a differencing technique that compares two versions of source code and detects the changes that are likely to impact requirements. In the second step, we build an approach to trace the relevant code changes to the requirements in order to identify the outdated ones.

RQ 3: How successful is our code-based approach in identifying outdated requirements? (Evaluation question) In RQ 3, we aim at evaluating the approach we designed for answering RQ 2. To answer this question, we apply the approach to two case studies and evaluate how effective it is for identifying the outdated requirements. The approach compares two versions of the source code and detects the relevant changes. These changes are then traced to the requirements. As a result, we obtain a list of requirements that are ranked according to their likelihood of being outdated.

In the evaluation, we look at how good the approach is in identifying the relevant code changes and how good it is in identifying the requirements that are impacted by these changes.

1.3.2 Research Direction II: Using Changes in High-Order Tests to Get Hints about Changes in Requirements

RQ 4: What information about requirements change can we obtain from changes in high-order tests? (Exploratory question) As high-order tests (acceptance tests, function tests and system tests) are meant to test the external behaviour of the system, changes in these tests are likely to mean change in the system behaviour. Additionally, many of these tests derive directly from the requirements specification, as they are meant to check that the implemented system satisfies the specified requirements. Therefore changes in high-order tests can be a relevant indicator for changes in requirements.

In RQ 4, we aim at exploring the relation between changes in high-order tests and changes in requirements. To do this, we describe the possible effects of various types of changes on both the tests and the requirements. The types of change we consider are the following: reductive maintenance, corrective maintenance, enhancive maintenance and performance maintenance.

RQ 5: How can we get automatic hints about changes in requirements based on changes in high-order tests? (Design question) RQ 5 builds upon the analysis done in RQ 4 of the effects of different types of changes on the tests and on the requirements. With this question, we aim at constructing a

new approach that uses the information obtained from changes in high-order tests to generate hints about the changes that need to be applied to the requirements. The hints that we consider in this question are the following: (1) identify the requirements that need to be changed or removed and (2) detect new requirements and provide information about them.

1.4 Research Methodology

The research methodology used in this thesis was inspired by the conceptual framework of Wieringa and Heerkens about *world problems* and *knowledge problems* in the *engineering cycle* [WH06]. World problems relate to how we think the world should be compared to how it actually is, while knowledge problems relate to our lack of knowledge about how the world is. Therefore, engineering problems are world problems and research problems are knowledge problems. In practice, there is mutual recursion between knowledge problems and engineering problems, as engineers need first to gather knowledge about the status of the world before trying to change it. Our work follows the typical engineering cycle, which is composed of the following steps that all belong to the world problems domain: (1) problem investigation, (2) solution design and implementation and (3) solution validation.

To address the problem investigation (world problem 1), we needed to answer the following knowledge question: "what are the existing approaches for supporting the maintenance of the requirements specification and what are the main limitations of these

approaches?". We answered this question using a literature review where we explored existing requirements update approaches and their limitations.

To design and implement a new solution for automatically identifying outdated requirements (world problem 2), we needed to answer the following knowledge question: "What relations are there between changes in the source code and changes in the requirements?". To address this question, we conducted an exploratory case study on an open-source software system and explored the effect of different code changes on the requirements.

To evaluate our solution (world problem 3), we conducted two case studies that aim at answering the following knowledge question: "How good is our approach in identifying outdated requirements?". The evaluation was conducted based on the goal-question-metric method [Bas92], where we first define the goal of the evaluation, then we define a set of questions to assess the goal and finally we define the metrics needed to answer the questions in a measurable way.

1.5 Roadmap and Chapter Summary

This thesis is composed of a set of peer-reviewed papers, where each paper is a chapter. Each of the papers includes some components that contribute to the achievement of the thesis goal. In this section, we present the contribution of each of the chapters and

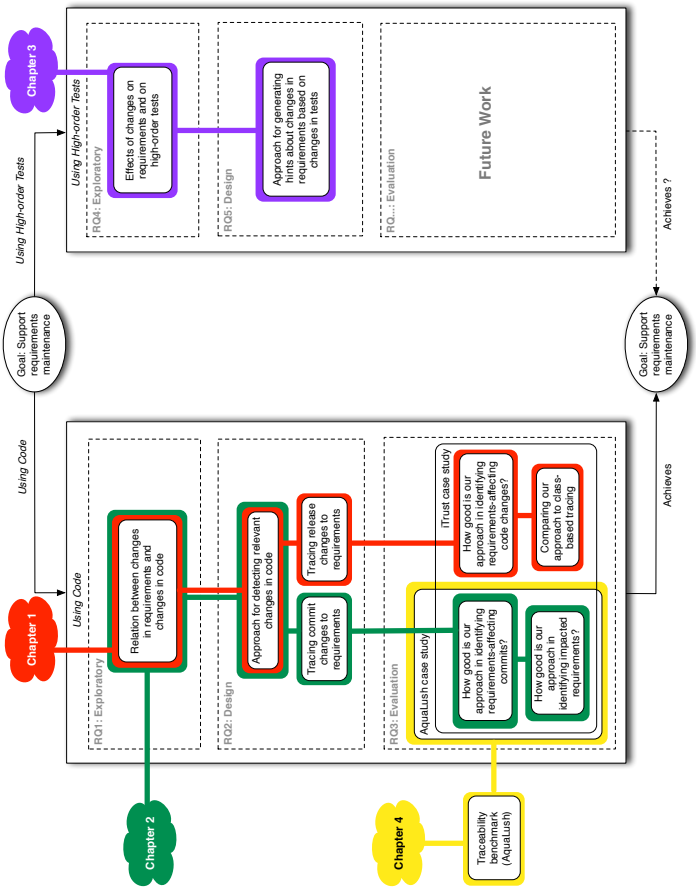

Figure 1.3: Research questions and answers covered in thesis chapters

its relation to the research questions presented in Section 1.3. An overview of the chapter content and their relation to the research questions is presented in Figure 1.3.

In Chapter 2 of the thesis, we present heuristics about the relations between changes in requirements and changes in code (RQ 1), then we present an approach for identifying the relevant changes in the code and for tracing the changes to the requirements in order to identify the affected ones (RQ 2). To evaluate the approach, we apply it to the iTrust case study and report the results (RQ 3). Chapter 3 is an extension of Chapter 2. In Chapter 3, we re-use the heuristics identified in Chapter 2 as well as the approach for identifying relevant changes. On this basis, we develop a new tracing technique that is adapted for commits (RQ 2). We apply the new approach to two case studies (AquaLush and iTrust), and report the results (RQ 3). The AquaLush case study, which we extended and used for the evaluation, is also a contribution by itself as it can be used as a benchmark for traceability. We present the details of the benchmark in Chapter 5. In Chapter 4, we explore the relations between changes in code and changes in high-order tests (RQ 4) and we present the approach for using tests to generate hints about changes in requirements (RQ 5).

In the remainder of this section, we summarize the content of each of the chapters.

1.5.1 Chapter 2: Identifying Outdated Requirements Based on Source Code Changes

Chapter 2 is about our code-based approach for identifying outdated requirements. The approach is built based on observations we made in an exploratory case study, where we identified relations between changes in requirements and changes in source code. The approach is composed of three steps. First, two versions of the source code are compared to each other in order to identify the relevant changes. The comparison is done using a differencing technique that we developed and which detects the changes in the code that are likely to impact requirements. Then a set of keywords describing each of the relevant changes is extracted. We extract these keywords from the code itself. Finally, the keywords are traced to the requirements specification in order to identify the parts that are impacted. The output of this approach is a ranked list of the requirements that are likely to be impacted for each changed class in the code. To evaluate the approach, we applied it to a case study of a software system for managing medical data (iTrust). To run the experiment, we developed a tool that compares two versions of source code, identifies the relevant changes and extracts keywords describing them. For the tracing, we used an existing tracing tool (Retro [HDS+07]) that is based on information retrieval. The results of the experiment were positive. In fact, the outdated requirements were among the five first requirements in the list for more than 50% of the cases, and it was the first one in the list for 10 out of 26 cases. The approach

also allowed identifying the relevant changes in the code with a precision of 79% and a recall of 85%.

1.5.2 Chapter 3: An Automated Approach to Identify the Requirements Impacted by Code Commits

In Chapter 3, we extend the approach presented in Chapter 2 for identifying outdated requirements. In this extension, the approach is changed so that it works after each code commit. The new approach identifies whether a commit is requirements-affecting or not. If the commit is requirements-affecting, then our approach generates a list of the requirements that are likely to be impacted. The list is obtained by combining the lists of affected requirements obtained for each of the changed classes. The final ranks of the requirements depend on the intermediate ranks obtained for each of the classes as well as on how many times the requirements appear in the intermediate ranks.

To evaluate the approach, we applied it to two case studies: (1) iTrust, which was also used to evaluate the previous version of the approach and (2) AquaLush, which is a software for managing an irrigation system. For both case studies, 70% to 100 % of the outdated requirements were identified within a list that includes less than 20% of the total number of requirements in the specification.

1.5.3 Chapter 4: An Automated Hint Generation Approach for Supporting the Evolution of Requirements Specifications

In Chapter 4, we present an approach that uses changes in tests to identify changes in requirements. The approach is based on the following two observations:

- Tests are usually maintained with the code.
- High-order tests, which are meant to test the external behaviour of the system, are usually derived from the requirements, and thus changes in these tests are likely to relate to changes in requirements.

As high-order tests are usually derived from requirements, linking the requirements with the tests should be an easy task. Therefore, high-order tests can be used as an intermediate between requirements and code and thus can be used to support the maintenance of the requirements specification.

To develop the approach, we consider different types of changes that affect the source code and analyse the effect of each of these changes on the requirements and on the high-order tests. The approach uses the changes in high-order tests to identify affected requirements and to obtain hints about new requirements. To identify outdated requirements, we define four rules that evaluate the likelihood of a requirement to be impacted based on its relation to the changed tests. In the case of addition of new requirements,

we use information from the added tests and from their execution traces to obtain information about the new requirements and their relation to old requirements.

To use the test-based approach, a complete and maintained set of high-order tests as well as traceability links between the tests and the requirements are required.

1.5.4 Chapter 5: Towards a Benchmark for Traceability

Data sets and case studies to be used for evaluating traceability techniques are scarce. To address this problem, the traceability community is currently engaged in developing benchmarks for traceability. The advantage of benchmarks is that they facilitate the evaluation of new techniques and the replication of experiments. The benchmark we present in Chapter 5 contributes to the extension of the data set being built by the traceability community.

Our benchmark has two main characteristics. First, it includes a rich set of artifacts from different phases of the software development lifecycle. Second, it includes end-to-end traceability links starting from the requirements to the design and architecture, then to the code and finally to the tests. The benchmark was built based on an existing case study of a software for an irrigation system: AquaLush [Fox06]. AquaLush was originally developed as an illustrating example for a book about software

design. Therefore it includes a rich set of high-quality artifacts. We extended the existing data by adding new artifacts and by defining the traceability links between the elements.

As a validation of our benchmark, we run the Retro tool using the data of our benchmark and compared the results we obtained to those obtained by other researchers using the same tool on different data sets. The results obtained with AquaLush were in the same range, in terms of precision and recall, as those obtained from the other data sets.

1.6 Contribution

The main contributions of the thesis are:

- An analysis of the relationship between changes in source code and changes in the external behaviour of the system,
- A new approach for identifying outdated requirements based on source code changes,
- An approach for generating hints about requirements change based on changes in high-order tests,
- A benchmark for traceability with a rich set of artifacts and end-to-end traceability linking.

Chapter 2

Identifying Outdated Requirements Based on Source Code Changes

Original publication:
Identifying Outdated Requirements Based on Source Code Changes
E. Ben Charrada, A. Koziolek, and M. Glinz
International Requirements Engineering Conference 2012

Abstract

Keeping requirements specifications up-to-date when systems evolve is a manual and expensive task. Software engineers have to go through the whole requirements document and look for the requirements that are affected by a change. Consequently, engineers usually apply changes to the implementation directly and leave

requirements unchanged. In this paper, we propose an approach for automatically detecting outdated requirements based on changes in the code. Our approach first identifies the changes in the code that are likely to affect requirements. Then it extracts a set of keywords describing the changes. These keywords are traced to the requirements specification, using an existing automated traceability tool, to identify affected requirements. Automatically identifying outdated requirements reduces the effort and time needed for the maintenance of requirements specifications significantly and thus helps preserve the knowledge contained in them. We evaluated our approach in a case study where we analysed two consecutive source code versions and were able to detect 12 requirements-related changes out of 14 with a precision of 79%. Then we traced a set of keywords we extracted from these changes to the requirements specification. In comparison to simply tracing changed classes to requirements, we got better results in most cases.

2.1 Introduction

Requirements specifications are used by engineers for several maintenance-related tasks [dSAdO05], such as comprehending programs, getting the rationale behind the implementation, identifying critical parts in the system, and discussing changes with stakeholders. Therefore, losing the knowledge contained in the requirements specification hinders the maintainability of software systems and limits their capacity for evolution. Nevertheless, the requirements specification is often not updated when the software evolves [BR00] [LSF03] [GS05], because updating the requirements document, which might include hundreds or thousands of pages written in natural language, is still a manual task that requires a lot of time and effort. Therefore, engineers usually choose to apply changes to the implementation only and leave the requirements document unchanged, as observed by e.g. Lethbridge et al. [LSF03]. As a result, the specification becomes outdated and looses its value.

The existing approaches for keeping requirements up-to-date can be classified into two categories: *Normative approaches* require developers to update requirements first, before any code changes are made [HWP09]. However, these approaches suffer from large manual effort to update both artefacts. *Trace generation approaches* such as [HDS06] aim at generating traces between requirements specification and code in order to support developers when analysing the impact of a change in one artefact on the other.

However, these approaches aim at generating all traces and do not consider the context of a specific change.

The work we present in this paper aims at supporting the update of requirements specifications when software systems evolve. We assume a situation that is frequently encountered in practice: An engineer modifies an existing system by making changes to the source code. No traces between code and requirements exist. The engineer knows that he should update also the requirements specification. However, under the usual time pressure, he will only do this if it can be done with little additional effort. Our approach provides help at this point: By analysing the changes in the source code, we semi-automatically identify the requirements affected by these changes and need to be updated, hence.

Our approach is composed of two main components. The first component is a code differencing technique that focuses on identifying source code changes that are likely to affect requirements. The technique was built based on an exploratory case study, where we made several observations of how we can differentiate between such requirements-related changes in code and changes that are refactorings or bug fixes. The second component of the approach is a technique for extracting relevant keywords describing the identified change and its context. The keywords are extracted from the name of the changed elements in the code, their documentation and their call graph. These keywords can then be traced to requirements using any automated traceability tool in order to identify the requirements that are likely to be impacted by the change.

To validate our approach, we applied it to a case study of a health care system. In the first part of the validation we assessed the effectiveness of our approach for identifying requirements-related changes. In the second part, we evaluate the effectiveness of using change-related keywords for tracing to requirements instead of tracing the class directly. In the change identification part, our tool succeeded to detect 12 of the 14 requirements-related changes, while extracting considerably fewer changed classes than the normal Eclipse comparator (33 with our approach against 91 with Eclipse), thus providing less irrelevant information. In the second part, our tool was able to provide a better ranking of potentially outdated requirements than a class-based tracing approach.

Our approach is expected to help the developer first to identify the changes in the code that are likely to affect the external behaviour of the system and then to find the requirements that are related to them. The approach can either be used in a fully automated way, where the changes are directly traced to requirements or in a semi-automated way, where the user can filter out manually the changes that he thinks are not relevant before running the tracing. Even in the semi-automated configuration, the manual effort required from the maintainer is small. Automatically identifying outdated requirements will make the life of the maintainer easier as it will reduce the time and effort needed for performing the update. Thus it should also encourage engineers to maintain the requirements after each code release.

The contribution of this paper is a novel approach for semi-automatically identifying outdated requirements when software

systems evolve. This work mainly targets functional requirements. The approach contains two novel features: First, we identify possibly requirements-related changes in source code based on observations how requirements-related changes differ from refactorings and bug-fixes. Second, we propose to extract keywords for tracing only from the changed elements and their context, such as call hierarchy and containing code elements. Furthermore, we provide a prototypical implementation and validate it in a case study.

The paper is organized as follows. In Section 3.3.1, we describe an exploratory case study to characterize source code changes that likely affect functional requirements and to derive heuristics to identify such changes. Section 2.3 describes our approach to automatically detect outdated requirements based on changes in the code. Section 2.4 presents the validation of our approach in the case study. Finally, Section 2.5 discusses related work, Section 2.6 highlights issues for future research, and Section 2.7 concludes.

2.2 Exploratory Study: Identifying Relations Between Changes in Code and Changes in the System External Behaviour

Functional requirements usually describe the external behaviour of the system, therefore in this work we will assume that the changes

affecting the external behaviour of the system are also affecting the functional requirements. To know which types of changes in code are likely to affect the external behaviour of the system, we conducted a small exploratory study using a real software project, namely an open source project for a barcode reader called ZXing[1]. The research question of this study is

> RQ1: What heuristics can be used to identify source code changes that likely affect the external behaviour of the system?

We compared two versions of the source code and made observations about how to differentiate between changes that are likely to be related to changes in system behaviour and changes that are refactorings or bug fixes. In our exploratory study, we went through all the changes between the versions 1.6 and 1.7 of ZXing manually and studied how requirements-related changes differ from refactorings and bug fixes. For some randomly selected packages, we studied the changes in detail and counted the frequencies. In this section, we present the six observations we made and what heuristics for identifying relevant changes we can derive from them.

Observation 1: Changes in methods bodies are in most cases related to refactoring and/or bug fixes Changes in methods bodies are among the most frequent changes that are applied to the code. However, they are not the most important

[1] http://code.google.com/p/zxing/

ones in terms of affecting the external behaviour of the system. In fact, most of the changes observed in the body of methods in the explored project where minor changes that relate to either refactorings or bug fixes. We also noticed that the few changes in methods bodies that related to additions or extensions of features to the system came along with additions of new elements (e.g. new classes, methods or fields). For example, in the packages that we chose to confirm the observation, we identified 33 changes in methods bodies. 23 of these changes were due to refactorings and bug fixes (the majority, 19, being refactorings) and 6 changes were related to the additions of new features. For the other 4 changes we could not guess what was the intent of the developer behind it. All the changes related to feature extension came along with additions of new elements in the code. Based on these observations, we derive the heuristic to ignore the changes in methods bodies.

Observation 2: Additions of new elements (classes; methods; package; fields) are usually related to the addition or extension of features We noticed that extension and addition of features are in most cases implemented through an addition of new elements in the code, where the names of these added elements usually reflect the implemented feature. It is important to note that there were some cases where the added element only relates to some implementation details. Therefore it is wrong to assume that *all* additions are extensions. However, we still can derive the heuristic that additions of new elements (additions of packages, classes, methods and/or fields) likely affect the external behaviour of the system.

Observation 3: Additions and removals of elements having similar names are usually rename operations When using normal differencing tools, renames are detected as an addition and a removal of two different elements. This can be very misleading as addition of elements is likely to relate to feature extension while renames are simple refactorings. When exploring the ZXing project, we noticed that in many cases, the new name is very similar to the old one (e.g. the field PDF417 was renamed to PDF_417). Therefore renames could be identified by computing the similarity between the name of the deleted and the name of the added one.

Observation 4: Changes in methods signature are usually related to refactoring Changes in methods signatures (other than renaming the method) were among the frequent changes that we observed when exploring the ZXing project and were in most cases related to refactoring. These changes can affect the visibility of the method (public, private, etc.), its return type (e.g. int, boolean, object...) and/or the parameters of the methods (e.g. the type of the parameters). In the packages we used for confirming the observation, all of the signature changes were due to refactoring. We derive the heuristic to ignore such signature changes.

Observation 5: Changes in private elements can affect the external behaviour of the system When starting our exploratory study, we were expecting to find that changes in public elements are likely to affect the external behaviour of the

system while private elements will only relate to implementation details. However, this was not the case in the ZXing project, as there were many changes in private elements that affected the external behaviour of the system. Therefore, we include changes in private elements when looking for changes affecting the system behaviour.

Observation 6: Additions of several methods having the same name are usually related to the same feature In many cases, we noticed that several methods having exactly the same name but different parameters where added to a class. In almost all cases, these methods related to the same feature and had similar behaviour. Therefore we derive the heuristic to consider and analyse only one of the added methods instead of considering them all.

2.3 Approach for Identifying Outdated Requirements

This section presents our approach for identifying outdated requirements. The approach consists of 3 steps (see Figure 1). First we identify the changes that are likely to affect the external behaviour (Section 2.3.1) based on the observations and heuristics presented in the previous section. Then, for each change we extract a list of keywords from the names of the changed elements, their documentation and their call hierarchy (Section 2.3.2). Finally we trace the

extracted keywords to the requirements specification in order to identify outdated ones (Section 2.3.3). To do the tracing automatically, we use an existing traceability tool based on information retrieval.

2.3.1 Identifying Relevant Changes

The first step of our approach is to detect the differences between the two versions of the source code and identify the relevant changes, i.e. changes that are likely to relate to a change in the external behaviour in the system and thus to relate to changes in requirements. Ideally refactorings, bug fixes and changes in the code documentation should not be detected by the approach. For achieving this goal, we developed a simple comparison technique which builds upon the heuristics presented in Section 3.3.1. Our approach is targeting code written in object-oriented programming languages.

We suppose that the code is composed of the following elements: packages, classes, methods and fields. Our comparing strategy consists in focusing on only two types of change: addition and removal of elements. First, we compare the packages in both versions and detect those that have been added or removed. The comparing is done based on the name only: a package is considered as added to the new version (respectively removed from the old version) if there is no other package in the old version (respectively the new version) that has the same name. Second, we go through each of the packages that appear in both versions of the code, and

compare the classes it contains. Here again we do the comparing based on the class name to detect added and removed classes. Third, we go through each class that appears in both versions and compare the methods and fields it contains. By detecting all added elements, this approach is detecting the main code changes that relate to feature addition/extension (observation 2). At the same time, the approach ignores many of the changes that are usually related to refactoring such as changes in method bodies (observation 1), changes in element signatures (observation 4) and changes in documentation.

As our comparing technique is based on element names only, renaming is detected as a simultaneous addition and removal of two elements. To filter out renames, we compare the names and the call hierarchy (for classes and methods) of the added and the removed elements. If the added and deleted elements belong to the same parent element (e.g. two fields belong to the same class) and if they have similar names, then the change is considered as a rename (observation 3) and is ignored by our approach. In the case of methods and classes, we also explore the call hierarchy of the elements: if the added and the deleted element have the same call hierarchy, then it is a rename. The similarity between the names of elements can be calculated in several ways such as using the Levenshtein distance [Lev66].

The output of this step is a set of source code elements that have been added and deleted and which are supposed to be requirements-related.

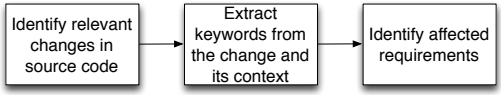

Figure 2.1: Approach for identifying requirements affected by change

Table 2.1: Elements used for extracting keywords for each type of change

Changed Element	Names	Documentation	Call Hier.
Package	package, sub-classes	none	No
Class	class, sub-methods, sub-fields	class	Yes
Method	method, parent class	method, parent class	Yes
Field	field, parent class	parent class	No

2.3.2 Extracting Keywords

In this part, we extract a list of keywords describing the change. We consider three sources of keywords (see Table 3.1): (1) the names of elements related to a change in the code (more details are given in the next paragraphs), (2) the documentation of elements and (3) their call hierarchy. The list is then reduced by filtering out irrelevant keywords and by grouping keywords. We detail each of the steps in the reminder of this section.

Names of the Change-Related Elements

The name of an element in the source code usually reflects its intended function. Therefore, in our approach, we use the names of the elements that relate to a change to extract the keywords

describing the change. The change-related elements include the name of the element that has been added/deleted in the code and the names of its parent or its sub-elements. Column 2 of Table 3.1 presents the elements that we consider as change-related for each type of changed element. For example, if a new class is added, then we consider the name of the class and the names of the methods and fields it contains.

As naming conventions differ from one language to another, the approach for extracting the keywords depends on the used language. For a concrete example, we consider the camelCase convention, which is used in several programming languages (e.g. Java and .NET). For the camelCase convention, the names are split according to the position of capital letters in the name, so that camelCase is split into "camel" and "case". If several subsequent capital letters appear in the name like generateHTMLReport, then these letters are considered as one keyword so that the results is "generate", "html" and "report". If a keyword is composed of one letter only or is a special character (not alphanumeric) then it is deleted.

Call Hierarchy

When a new element is added, considering only the names of the element and its parent/children might not be enough to determine the context of the change. Therefore, we need more information about when such an element is used. To get such information, we look at the call hierarchy of the element. We consider the call hierarchy when the added/removed element is a method or a

class. By call hierarchy we mean all the methods/classes that are invoking the considered method or those invoking the a constructor of the considered class. The invocation can be either direct or via other methods/classes. As the call hierarchy of an element can be large, we should not go back too far in it. We expect that going back by one, two or three levels in the call hierarchy should be enough to gather relevant information about the context of the change.

To extract keywords from the elements identified in the call hierarchy, we use the same approach that we described in the previous section for extracting keywords from names of elements.

Documentation

The documentation of elements is also a valuable source of information that we consider in our approach. We add the terms in the documentation of important elements to our list of keywords. The third column of Table 3.1 presents which documentation we consider for each type of change.

Filtering and Grouping

Before tracing the keywords, we filter out irrelevant keywords and group changes together. Applying the keywords extraction approach is likely to generate many irrelevant keywords such as keywords relating to implementation details (set, get, string, etc.),

very general keywords in the project (e.g. the name of the project), or stop words that might appear in the documentation of elements (a, the, that, etc.). Having irrelevant keywords in the list might have negative effect on the tracing of changes to requirements. Therefore it is important to filter out as many of these irrelevant keywords as possible. Filtering can be done in several ways. One possibility is to manually prepare a list that combines the most common words in the code and a stop word list. A more sophisticated way to construct the list is to build it automatically based on the frequency and appearance of keywords in the considered source code.

Considering each change separately will results in a very high number of changes, where each change might not be relevant on itself. Therefore it is important to group changes together. One possible way to group changes is to consider all changes affecting a class as a single change, where the keywords extracted from all the changes in that class are grouped together. Although this grouping might not be the most efficient one, it should reduce the number of changes to be traced to a reasonable number for medium size projects.

2.3.3 Identifying Affected Requirements

The last step of our approach is meant to identify the requirements that are affected by the change based on the keywords we extracted and grouped in the previous steps. This can be done by using IR-based techniques to trace the extracted keywords to requirements.

As most IR-based tracing techniques perform similarly [OGPDL10], the selection of a concrete technique to use is not such important. The requirements identified by the tracing tool are the ones that are likely to be affected by the change and thus are the ones that the maintainer should review.

2.4 Evaluation

This section describes the evaluation of our approach in a case study. Section 2.4.1 describes the prototypical tool we implemented to automate our approach. Section 2.4.2 then describes the case study system, the iTrust Health care project.

The evaluation consists of two studies designed to answer the following research questions. In the first part, described in Section 2.4.3, we evaluate the effectiveness of our approach for identifying requirements-related changes (relevant changes), thus validating the first step of our approach as shown in Figure 2.1. The evaluation question is

> EQ1: How effective are the proposed heuristics for differentiating between changes which impact requirements and those which do not?

In the second part, described in Section 2.4.4, we evaluate the effectiveness of using change-related keywords for tracing the changes to requirements instead of tracing classes directly, thus validating the second and third step of our approach as shown in Figure 2.1. The evaluation question is

> EQ2: Does our approach of change-based tracing give better results than the class-based tracing? If yes, how much?

The metrics to assess the quality of the change identification (EQ1) and the quality of trace results (EQ2) are presented in sections 2.4.3 and 2.4.4.

Finally, in Section 2.4.5 we discuss the significance of our findings and threats to validity. We do not combine the two parts in an end-to-end validation because there are no existing approaches to meaningfully compare to, as discussed in more detail in Section 2.4.5.

2.4.1 Tools

In this section we present the two tools we used to run our experiment. The first tool is a prototype that we developed based on the first and second steps of our approach. This tool was used to compare and identify the relevant changes between two versions of source code and to extract the keywords related to these changes. The second tool is an information retrieval based traceability tool called Retro. We used Retro to trace the keywords found by our prototype to the requirements specification.

Prototype Our prototype (1) compares two versions of code and identifies the requirements-related changes and (2) extracts the keywords related to each of the changes. Although the comparing technique and the keywords extraction parts are not completely separated from each other in the implementation, we present each part separately for the sake of comprehensibility.

The comparing part was developed based on an existing Java library for comparing Java API called JDiff [Doa02]. We have chosen JDiff, as it has similarities with our comparing technique: JDiff detects the elements that are added or removed in the code, but does not detect changes in methods bodies. We adapted JDiff so that it ignores changes that are not relevant in our case, such as changes in elements signature. We also changed the default behaviour so that it also detects changes in private elements. To filter out rename operations, we implemented a comparator that compares element names and their call hierarchy: if an added element and a deleted one have similar names and/or have the same call hierarchy then the change is considered to be a rename and is ignored by the tool. The name similarity is computed based on the Levenshtein distance [Lev66].

For each of the changes identified by the comparing part, the tool extracts a set of keywords related to it as presented in Section 2.3.2. The tool contains a configurable list of keywords (stop word, project-specific common words) that are filtered out when building the list of keywords for each change.

The extracted keywords can then be reviewed by the user. There are two possible display configurations: either the keywords are

grouped by change, which is very fine-granular, or grouped by class.

Retro Retro (REquirements TRacing On target)[HDS+07] is an automated tool for generating traceability links among textual artefacts. Retro implements various IR-based techniques for link generation. Retro takes as input two lists of textual files: the high-level documents and the low-level documents and traces them to each other. The output of Retro is a list of candidate links that are sorted according to their relevance.

Retro includes other functionalities, which we did not use in our experiment, such as filtering links having relevance lower than certain threshold values, and entering analyst feedback to improve the generated links.

Configuration In this paragraph we specify the configuration we used in our experiment for each of the tools. For our prototype, we considered changes in both public and private elements, and we set the depth of the call hierarchy to two. We also used the keywords grouped by class. For the Retro tool, we used the default tracing method, which is the vector space retrieval with tf-idf (term frequency - inverse document frequency) term weighting.

2.4.2 Case Study

We used the iTrust Medical care project [MSW12] as a case study for the evaluation. iTrust, is a tool for managing medical data

and has been developed for teaching purposes at the North Carolina State University. The tool has a wiki-based requirements specification that includes functional requirements, non-functional requirements, a glossary, a set of global constrains (e.g. programming language, coding standards, etc.) and a section dedicated for specifying the data format for the input fields [WXM+]. The tool is a web application that is developed using a combination of Java code and Java Server Page. A new version of the code, which is maintained by students in software engineering, is released every semester.

For the case study, we only considered the functional requirements, which are specified in the form of fine-grained use cases. There are around 40 use cases in total. Figure 3.3 shows an example use case of the system.

For the code, we only considered the part written in Java as our prototype only works on Java code. We used versions 10 (release date: August 18th, 2010) and 11 (release date: January 7th, 2011) of the source code. To obtain the requirements that correspond as much as possible to each of these releases, we choose a wiki version from a date that is after the code release and before the beginning of the following semester. The reason is that, after the release, the project owners do a cleanup and maintenance for the requirements based on the work done by the students. Therefore we consider the requirements specification as of September 3rd, 2010 (the "old requirements") for the source code version 10 (the "old code version") and the requirements as of February 7th, 2011 (the "new requirements") for the source code version 11 (the "new code version") [WXM+].

UC1 Create and Disable Patients Use Case

1.1 Preconditions: The iTrust HCP has authenticated himself or herself in the iTrust Medical Records system [UC3].

1.2 Main Flow: An HCP creates patients [S1] and disables patients [S2]. The create/disable patients and HCP transaction is logged [UC5].

1.3 Sub-flows:

[S1] The HCP enters a patient as a new user of iTrust Medical Records system. Only the name and email are is provided. An email with the patient's assigned MID and a secret key (the initial password) is personally provided to the user, with which the user can reset his/her password. The HCP can edit the patient according to data format 6.4 [E1] with all initial values (except patient MID) (...)

[S2] The HCP provides the MID of a patient for whom he/she wants to disable [E2]. The HCP provides a deceased date (data format 6.4). An optional diagnosis code is entered as the cause of death.

1.4 Alternative Flows:

[E1] The system prompts the enterer/editor to correct the format of a required data field because the input of that data field does not match that specified in data format 6.4 for patients.

[E2] (...)

Figure 2.2: Example use case from iTrust requirements specification [WXM+], version of September 3rd, 2010

We manually compared the two versions of the code and identified the main changes that relate to the external behaviour of the system. To make sure that we did not miss any important change, we used the Java source compare in Eclipse, which identifies all textual changes (including addition/removal of spaces). Eclipse identified 91 changed classes.

We went through all of these classes and identified 14 different requirements-related changes, i.e. changes that should affect requirements. One requirement-related change can be scattered over several classes, so that each class contains a part of this change. The total number of classes that contain requirements-related changes is 31. Then we went through the requirements specification and identified all the use cases that are affected by each of these changes.

To check the completeness of our change list, we compared the old and the new versions of the requirements specification and looked for the requirements changed by the owners of the project. This comparison was challenging because of two reasons. First, we observed that both versions of the requirements specification do not perfectly match the respective code versions. Sometimes, the requirements specification listed a requirement that was not yet implemented in the respective code version. In other cases, the requirements specification was indeed outdated, so it did not reflect a recent behaviour-changing change in the respective code yet. Second, as we only consider the Java part of the source code, it is likely that we missed the changes that affect the JSP part only. This comparison was still helpful, as it helped us decide which uses cases should be updated for certain changes.

2.4.3 Study 1: Identification of Requirements-Related Changes (EQ1)

The goal of the first part is to evaluate how well the change identification step of our approach identifies requirements-related changes.

Experiment Design

We run our change identification step on the two iTrust source code versions. It reports a set of classes that are supposed to contain parts of the requirements-related changes.

To assess the performance of our approach, we measure its precision and recall. For the recall, we determine how many requirements-related changes are covered by the classes reported by our change identification step. Due to the scattering and tangling between requirements and code, a change in one requirement is likely to show up in several classes and methods in the code. For identifying whether a requirement is outdated, it is enough if one of the changed classes is reported. Therefore, a requirements-related change can be deemed covered if at least one class that contains a part of this change is reported. The recall measure is defined as the fraction of requirements-related changes covered by the retrieved classes.

For the precision, we determine how many of the retrieved classes are relevant for the requirements-related changes. The precision measure is defined as the fraction of retrieved classes that actually contain at least one part of a requirements-related change.

Results

Using our comparing tool, 33 classes were identified, covering 12 of the 14 requirements-related changes. Among the 33 identified classes, 26 actually contained parts of the 14 changes. The other 7 classes were simple refactorings. Thus, our approach achieved a precision of 26 / 33 = 79% and a recall of 12 / 14 = 85.7%.

To conclude the first study, we observe that our approach was able to find most requirements-related changes and to exclude the majority of irrelevant classes.

2.4.4 Study 2: Keyword Extraction and Tracing Results (EQ2)

In this part we evaluate how well our approach can identify affected requirements, i.e. use cases in this case study, based on the extracted keywords.

Experiment Design

We run the keyword extraction and tracing step for each of the 26 relevant classes from the previous part of the evaluation. Our choice for tracing the relevant classes only and not all of the 33 changed classes is explained in Section 2.4.5. The output of the two steps is a ranked list of candidate use cases for each class which

are suggested to be related to the requirements-related changes in this class.

To assess the quality of the rankings, we compare them to the true relation between the changes and the use cases, which we defined manually as discussed in Section 2.4.2. Each class we create a ranking for is related to one of the 14 requirements-related changes. Thus, our approach should report the use cases affected by that requirements-related change. The first two columns of Table 2.2 show this true relation of classes to use cases for the 14 requirements-related changes, which forms the ground truth for the keyword extraction and tracing step. For two classes, namely the `ActivityFeedAction` and the `ViewHelperAction`, a new requirement was introduced, so no use case could be matched in the requirements specification (marked NEW in Table 2.2). Let Z denote the set of considered classes and let U denoted the set of related use cases. We denote the true relation as $T \subset Z \times U$. A true link between a class c and a use case u is denoted $t = (c, u) \in T$.

To assess how well our approach performs compared to existing approaches, we compare our approach to a simple class-based tracing approach between the classes and the use cases of the requirements specification as a baseline, described in the following. In the class-based tracing, we use all the keywords in the file of the class as input to compute the similarity of that class with the use cases. The output of the class-based tracing is a ranked list of candidate use cases that are suggested to be related to the class in general. Thus, to compare the usefulness of our change-based

approach and the class-based tracing in the requirements update scenario, we compare how close the rankings produced by the two approaches are to the true relation described above.

The produced rankings do not necessarily rank all use cases, but only a subset. For statistical validity, we use a fractional ranking where tied use cases receive a fractional rank number that is the mean of the ranking positions they would receive in ordinal ranking. Let us denote a suggested ranking R_c of a subset of the use cases $U_{R_c} \subset U$ for a class c as a function $R_c : U_{R_c} \to \mathbb{R}$ so that the rank of a use case u is given by $R_c(u)$. Furthermore, let $R_Z = \{R_c \,|c \in Z\}$ denote the set of rankings suggested by an approach for all classes.

To measure the quality of the rankings, we use three measures, namely the the median rank, precision, and recall.

First, we measure the median rank of the true links in the rankings produced by each approach. For each class c, let the use cases that should be found be denoted $U_c = \{u \in U \,|(c, u) \in T\}$. Then, the median rank of the true links is $\tilde{R}_c(u)$ for $c \in Z, u \in U_c \cap U_{R_c}$.

Note that this calculation ignores situations where a true link is not contained in a ranking, i.e. where $U_c \cap U_{R_c} = \emptyset$. As we will see later, this calculation favours the class-based tracing approach, so we do not present a more complex median calculation that accounts for missing ranks by e.g. assigning a default rank at the end of the ranking.

Second, we measure precision and recall. The precision of an approach is the fraction of retrieved true links, and the recall is the fraction of true links that were retrieved. Because it is easier for developers if true links are suggested early in a ranking, we study the precision and recall at a cut-off rank n, i.e. only links retrieved at ranks lower than n are considered (cf. [Dom08, Sec. 4.9.3]). More formally, the true links suggested by an approach up to cut-off rank n is

$$cutTrue(R_Z, T, n) = \{(c, u) \,|(c, u) \in T, R_c \in R_Z, R_c(u) \leq n\}$$

The number of all links suggested by an approach up to rank n is

$$cutAll(R_Z, T, n) = \{(c, u) \,|R_c \in R_Z, u \in U_{R_c}, R_c(u) \leq n\}$$

Then, the precision at n is

$$precision(R_Z, T, n) = \frac{|cutTrue(R_Z, T, n)|}{|cutAll(R_Z, T, n)|}$$

and the recall at n is

$$recall(R_Z, T, n) = \frac{|cutTrue(R_Z, T, n)|}{|T|}$$

To study how early relevant links are suggested by the approach, we increase the cut-off rank n from 1 to half the number of use cases to create a precision-recall graph [Dom08, Sec. 4.9.3].

Table 2.2: Ranks for classes. A cell is marked grey if one of its ranks is more than 5 ranks better than the rank of the other approach. A rank is underlined if it is between one and four ranks better than the rank of the other approach.

Id	Class	Affected Use Cases	Change-based Tracing	Standard Class-Based Tracing
1	ActivityFeedAction	NEW		
2	AddRemoteMonitoringDataAction	UC34	1	2
3	ApptDAO	UC22	29	1
4	DAOFactory	UC15, UC4	5, 30	21, 34
5	EditApptAction	UC22	3	3
6	EditOfficeVisitForm	UC11	4	1
7	EventLoggingAction	UC5	1	3
8	HealthData	UC10	7	14
9	LoginFailureAction	UC3	1	1
10	OfficeVisitDAO	UC11	4	1
11	OverrideReasonBean	UC15	28	30
12	OverrideReasonBeanLoader	UC15	11	29
13	OverrideReasonBeanValidator	UC15	23	34
14	PrescriptionBean	UC37, UC15	21, 1	16, 12
15	ProfilePhotoAction	UC4	7	8
16	ProfilePhotoDAO	UC4	33	37
17	ProfilePhotoServlet	UC4	NA	NA
18	ReasonCodesDAO	UC15	3	7
19	RemoteMonitoringDAO	UC34	1	1
20	RemoteMonitoringDataBean	UC34	1	5
21	RemoteMonitoringListsBeanLoader	UC34	1	26
22	TelemedicineBean	UC34	1	NA
23	TransactionDAO	UC5	2	4
24	UpdateReasonCodeListAction	UC15	1	1
25	ViewHelperAction	NEW		
26	ViewMyRemoteMonitoringListAction	UC34	1	7

Results

Table 2.2 shows the resulting ranks of the correct use cases as produced by our change-based approach and the comparison class-based approach. In three cases, the true link was not retrieved at all by an approach, which is marked by NA in Table 2.2. We observe that our tool performs better in 16 of 26 cases and even considerably better (i.e. 5 or more ranks better) in 7 cases. The class tracing performs better in only 4 cases, two of which are considerably better. In 5 cases, the approaches perform equally well.

When considering the cases in which our approach performed considerably worse, we detect two problems. For use case 23 for class `ApptDAO`, we find out that the problem comes from the fact the developers used abbreviations ("appt" for "appointment"), therefore our tool could not trace it to the appointment use case. However, class-based tracing was better because there was a message that should be displayed to the user in the body of the method and which contains the keyword appointment in it. For use case 37 and class `PrescriptionBean`, we observe that the addition of an ORC (an overriding reason code) was better ranked by the class-based tracing than by our approach (16 to 21). We found that the bean was only called from JSP classes, which were not considered in this study. The lack of a call hierarchy might have hindered our approach in this case. However, note that both approaches did not perform well and produced a high rank.

The median rank of the correct use case in our approach is $\tilde{x} = 4$, while it is $\tilde{y} = 7$ for the class-based approach. In fact, our approach

produces lower (i.e., better) ranks at a significance level of 0.05. We use a one-tailed Wilcoxon Signed Rank test with continuity correction as performed by the R statistics tool [R D10] because it is applicable to ordinal scales and we assume that the differences $x - y$ are independently distributed [Low12]. Our null hypothesis H_0 is that the difference in ranks $x - y$ is symmetric about 0 or larger. H_0 is rejected with $p = 0.013$.

Figure 2.3 shows the results for precision and recall at cut-off ranks n for $1 \leq n \leq 15$. We observe that our approach performs better with respect to both precision and recall. For example, if only the first returned use case is considered ($n = 1$), the class tracing has a precision of 0.23 and a recall of 0.23 while our approach has a precision 0.38 and a recall of 0.38, which is an improvement of 66% [2]. Furthermore, for a fixed recall value, the precision of our approach is at least twice as good as for the class-based approach.

To summarize, we observe that our approach was able to find better results than the class-based approach. Thus, for the second study we conclude that in this case study, a hint generation for outdated requirements based on keywords extracted from the change and its context is more useful than using class-based tracing for identifying outdated requirements.

[2] Note that precision and recall coincidently have the same value, as the number of classes is 26 and the number of true links is 26, too.

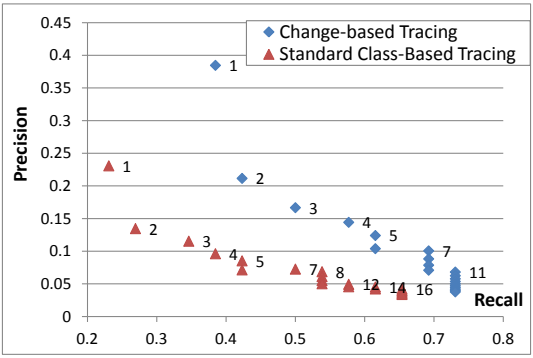

Figure 2.3: Precision / recall at n for ranks smaller than 15, the respective cut-off rank n is annotated to the data points.

2.4.5 Discussion

In this section we discuss the significance of our findings (Section 2.4.5), the threats to validity of our study (Section 2.4.5), and argue for the chosen two-part validation strategy (Section 2.4.5).

Approach Evaluation

The results obtained from both parts of the evaluation give a positive indicator about the relevance of the approach. In fact, our approach succeeded to cover most of the important changes and gave results that are in most of the cases better than those obtained by using class-based tracing. Additionally, the approach does not require much effort, as it can either be run fully automatically or with some user feedback. In the current evaluation,

the user feedback consists in removing the 7 classes that contained simple refactoring and which were still identified by the tool. An important characteristic of our approach is that it filters out much of the irrelevant information that hinder the tracing (e.g. the import packages in a class). It also considers the context of the change through the call hierarchy of elements. Another plus of our approach is that it is configurable (e.g. depth of call hierarchy, elements to be considered, etc.) thus it can be adapted to the characteristics of the used project.

The two main limitations of the approach are that (1) it can miss some relevant changes and (2) it does not generate links that are 100% correct. The first problem is related to the compromise between identifying as many relevant changes as possible, and identifying relevant changes only. If the approach is extended so that it covers more changes (e.g. changes in methods bodies) then it will detect more relevant and more irrelevant ones. The second limitation is a normal traceability problem: tracing is based on a pure textual analysis that only considers the keywords appearing in the traced documents without considering the intent behind it. Therefore, such a technique generates many false links between documents that are not related but contain similar keywords and misses the links between related documents that do not contain similar keywords.

Despite these limitations, the approach can still be very useful as it can support the maintainer during the update by suggesting him the requirements that are likely to be affected. Instead of manually analysing all requirements for whether they need to be

updated, the maintainer can focus on the requirements suggested by our approach first, which we expect to decrease the effort for updating the requirements considerably. To quantitatively assess how useful such tool is, we intend to conduct an experiment where we compare how efficient the update of requirements is with the help of our approach compared to manual requirements updated and standard-traceability-based requirements update.

Threats to Validity

External Validity As the evaluation was done based on one project only, the findings cannot be generalised to other types of projects. In fact, the results depend on several projects parameters such as the coding style, the structure and types of the requirements specification, etc. Nevertheless, the positive results obtained in the current evaluation indicate that the approach is beneficial for at least one type of software projects. Further evaluations will be conducted in the future, to assess the effectiveness of our approach on other types of projects.

Internal validity To ensure internal validity, we need to check whether the superiority of the results obtained by our approach over the class-based tracing approach is due to our approach. As we ran both experiments using the same project (same requirements specification) and the same traceability tool, the only parameter that has changed is the use of the change context instead of using the whole class. Therefore, the improvement in results can only

be due to our approach of extracting keywords from the change context.

A second threat to internal validity is related to the identification of the affected requirements (the ground truth). Identifying which requirements should be updated after a change can be a subjective matter. To avoid any bias, we did the manual identification of affected requirements before running any of the experiments.

Validation Strategy

We preferred an evaluation in two studies rather than doing an end-to-end validation of the approach, because we did not find other approaches performing the same task as ours that we could use to compare our approach to. We think that comparing our use of keywords extracted from the change context to class-based tracing in the second study is more relevant than presenting raw values for the whole approach that are not comparable to anything. Furthermore, comparing the whole approach to a class-based tracing approach would be unfair: We would need to trace all of the changed classes (91 classes) for the class-based approach, which would result in a very low the precision for the class-based approach because many of these classes were only refactored. This motivates our choice for tracing only the 26 relevant classes identified in Section 2.4.3 with both approaches.

2.5 Related Work

There are two categories of existing approaches to requirements update, namely (1) approaches prescribing to update requirements first and propagate the change to the code and (2) trace generation approaches generating traces between requirements specification and code.

First, the approaches for updating requirements specifications assume an ideal maintenance process where first the requirements are updated then the changes are propagated to the source code [HWP09] [EBM11]. In our approach, however, we consider the frequently recurrent case where only the code was updated while other documents, including requirements, were not modified. Here, our approach uses the changes that were done at the code level to support the update of requirements. In a previous work [BCG10], we propose a test-based approach for identifying requirements affected by change. A set of high-level tests and traceability links between these tests and requirements are used to identify the requirements impacted by each implemented change. In contrast, in our current work we do the analysis on the source code directly, so there is no need for the high-level tests and no need for any traceability links.

The main approach that is used for propagating changes between software artefacts, and which is also applicable for propagating changes between source code and requirements, is software traceability. Approaches for automatically generating traceability links

between software artifacts and for using traceability to manage and propagate change do exist [HDS06] [ACC+02] [CHCC03]. Our current work can be considered as a special traceability approach that focuses on tracing only the relevant code changes to requirements. There are two main differences between our approach and existing traceability approaches. First, our approach includes a feature for identifying the relevant changes that should be traced in the code. Second, we propose a new way for selecting the set of terms to be traced.

2.6 Future Work

There are two main parts for the future work. One part is about extending and improving the current approach. The other part is about the evaluation of the approach.

2.6.1 Extending the Approach

There are two techniques that we intend to incorporate in our approach. First, we plan to introduce weights for the keywords depending on their source. For example, keywords extracted directly from the change could be assigned a higher weight than keywords extracted from containing classes or from the call hierarchy.

Second, we plan to use the requirements specification to further improve the weighting of the extracted keywords. This can be done

by considering the occurrence of keywords in the requirements specification. We will use what we call *keyword specificity*, which we define as follows: if one keyword appears in several scattered parts of the requirements specification then it is not very specific and will either have a low weight or be filtered out completely.

2.6.2 Evaluating the Approach

To further evaluate our approach, we plan to apply it to different types of projects in the future. The goal of the evaluation is to find out for which type of projects our approach works best. Another part of the evaluation will be about the usefulness of our approach for requirements update. We will explore, using a controlled experiment, the effect of our approach on the maintainer's efficiency during the update and on the quality of the update.

2.7 Conclusion

In this work we present a new approach for identifying outdated requirements based on an analysis of code changes. Our approach has three main steps: First, the new and old versions of source code are compared and relevant changes are detected. Then a set of keywords describing the change is extracted from the names of the elements related to the change, their documentation and their call hierarchy. Finally related requirements are identified by tracing the extracted keywords to the requirements specification.

Our approach is meant to support the requirements update task when no predefined traceability links are available. Compared to a class-based tracing approach, our approach yields better results in terms of precision, recall, and the ranking of true links. The next step of our work would be to evaluate how useful our approach is for supporting the requirements update task.

Chapter 3

An Automated Approach to Identify the Requirements Impacted by Code Commits

Original publication:
Supporting Requirements Update during Software Evolution
E. Ben Charrada, A. Koziolek, and M. Glinz
Submitted to Journal of Software: Evolution and Process

Abstract

Updating the requirements specification when software systems evolve is a manual task that is expensive and time consuming. Therefore, maintainers usually apply the changes to the code directly and leave the requirements unchanged. This results in the requirements rapidly becoming obsolete and useless. In this paper,

we propose an approach that supports the maintainer in updating the requirements specification by identifying the requirements that are likely to be outdated after each code commit. Our approach works as follows. First, we analyse the changes that have been applied to the source code and detect if they are likely to affect the requirements or not. Second, we trace the requirements-impacting changes back to the requirements specification to identify the parts that need to be modified. The output of the tracing is a list of requirements that are sorted according to their likelihood of being impacted. Automatically identifying the parts of the requirements specification that need to be changed reduces the effort needed for keeping the requirements up-to-date and thus makes the task of the maintainer easier. When applying our approach in two cases studies, 70% to 100% of the outdated requirements were identified within a list that includes less than 20% of the total number of requirements in the specification.

3.1 Introduction

When maintaining and evolving software, an up-to-date requirements specification provides much knowledge about the software that is very useful for supporting several maintenance and evolution tasks. For example, the requirements include the rationale behind the implementation, which can support program comprehension and which also prevents undoing important decisions. Additionally, the requirements are usually written in natural language and thus can be used to discuss changes with stakeholders who are not from the software engineering domain. Therefore, losing the information contained in the requirements specification hinders the maintainability of a software system and leads it to eventually enter the *servicing stage* [BR00] where only minor changes can be applied to them.

In practice, however, requirements are usually not updated when software systems evolve [LSF03] [BR00] [GS05]. This is mainly because updating requirements is still a manual task that is very expensive and time consuming. In fact, the maintainer has to go through the whole requirements specification, which can include hundreds or thousands of pages, and find the parts that need to be changed. Therefore, maintainers usually apply changes to the code directly, but do not update the requirements specification as observed by Lethbridge et al. [LSF03], for example. Consequently, the requirements specification rapidly becomes obsolete and useless.

The goal of this work is to support the maintainer in keeping the requirements specification up-to-date when software systems evolve. We propose a new technique that automatically identifies the requirements that are likely to be outdated based on the changes applied to source code. Our approach detects the source code changes that are likely to affect the requirements and traces these changes back to the requirements specification in order to identify the parts that need to be modified. Tracing is automatic and does not require any manually created and maintained traces. Currently, our approach is limited to functional requirements, which are the dominant category of requirements in most systems.

To evaluate our approach, we apply it in two case studies: AquaLush and iTrust. For both case studies, our approach succeeded to detect most of the code changes that affect the requirements and succeeded to ignore irrelevant changes such as bug fixes and refactoring. When tracing the changes to the requirements specification, our approach identified all impacted requirements while filtering out 84% of the non-impacted ones for the AquaLush case study. This reduces the number of requirements that the user has to look at considerably. For the iTrust case study, our approach succeeded to identify 72% of the outdated requirements while filtering out 81% of the non-impacted ones. As our approach considerably reduces the number of requirements the maintainer has to look at to find the outdated parts, we expect it to encourage maintainers to update the requirements specification regularly.

This paper is an extension of an existing conference paper [BCKG12]. In this extension, we have three new contributions. First, we

changed the scope of our approach: before, the approach was meant to be used after each code release, but now it is used after each commit (or evolution task [NDCAC11]). Second, we have extended our tracing technique so that it merges the traces obtained from various classes in one final ranked list using a new scoring technique. Finally, we did an additional evaluation of the approach on a new case study (AquaLush), and we re-evaluated the new tracing technique (with the scoring) on the iTrust case study, which was used in [BCKG12].

The rest of the paper is organised as follows. In Section 4.3, we present an overview of the approach and its usage. In Section 3.3, we present the first step of the approach, which aims at identifying whether the changes of a commit are relevant or not. In Section 3.4, we present the second part of the approach, which consists of extracting the keywords for each change and tracing them to the requirements. In Section 3.5, we present the tools implementing our approach. In Section 3.6, we present the evaluation of our approach in two case studies: AquaLush and iTrust. We discuss the results of our approach in Section 3.7. Then, we present the related work about requirements update and software traceability in Section 3.8. The conclusion and future work are presented in Section 3.9.

3.2 Idea and Approach Overview

Among the various software artifacts that are created during the development of a software system, the source code is “the” arti-

fact that is changed whenever a change in the software system behaviour is needed. This is because no change in the system behaviour happens if the code is not changed. Implementing a code change requires an impact analysis that is done at the code level to identify all the parts that need to be modified. Our idea is to build an approach that takes advantage of the impact analysis that is done at the code level to automatically identify the impacted parts in the requirements.

From the maintainer's point of view, our approach is automated and fits to the classical maintenance process, where the maintainer commits the code after implementing each software evolution task. An overview of the maintenance process is presented in the activity diagram in Figure 3.1. After implementing a code change and committing the change to a version control system, the changes are automatically analysed to detect whether they affect requirements or not. If no requirements-affecting changes are detected then nothing is displayed to the maintainer. If, however, requirements-affecting changes are detected, then these changes are traced to the requirements specification and the related requirements are displayed to the maintainer. The maintainer can then edit these requirements, update them and save the changes. On the one hand, this approach can save much effort to the maintainer, who does not need to go through the whole requirements specification any more to find the outdated elements. On the other hand, it encourages the maintainer to update the requirements right after modifying the code.

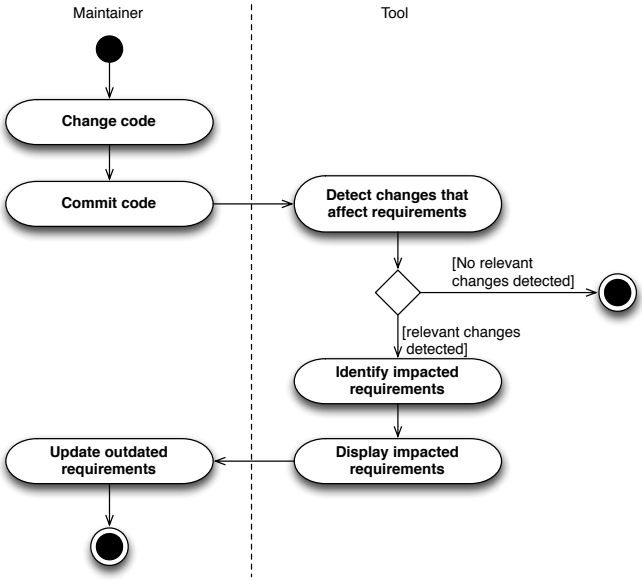

Figure 3.1: Activity diagram of the maintenance process using our approach

From the implementation perspective, our approach is composed of three steps: (1) identifying the relevant changes in the commit (**Differencing step**), (2) identifying the requirements that are impacted by the changes (**Tracing step**) and (3) displaying the impacted requirements to the user (**Displaying step**).

Differencing step The goal of the differencing step is to detect whether the commit includes requirements-affecting code changes or not. In the rest of the article, we use the term *relevant changes*

to refer to the code changes that affect the external behaviour of the system and thus are likely to affect its functional requirements. Changes that do not affect the requirements, such as refactoring and bug fixes are considered as *irrelevant* and are thus ignored by the approach. The challenge in this part is to find an *automated* way to detect whether a code change is relevant or not. To address this challenge, we conducted an exploratory study where we explored the relations between code changes and changes in the external behaviour of an open source software system (See Section 3.3.1). Based on the study, we came up with a set of heuristics about the effect of code changes on the external behaviour. We then used these heuristics to develop a differencing approach that detects only code changes that are likely to affect requirements. More details about the exploratory study and the concrete implementation of the differencing part are presented in Section 3.3.

Tracing step The goal of this step is to trace the relevant changes that were identified in the previous step to the requirements specification in order to identify the requirements that are likely to be impacted. The challenge in this step is to find a tracing technique that is *effective*. The tracing technique we propose is composed of two parts. The first part aims at gathering relevant keywords about the change and its context, by extracting terms from the changed elements in the code. In the second part, the keywords are traced to the requirements, and a list of impacted requirements is generated. Each requirement in the list is associated to a value that represents the likelihood of the

requirements to be impacted by the change. As mentioned above, we do not require any manual traces. The tracing step is detailed in Section 3.4.

Displaying step The goal of the displaying part is to display the results in a convenient way that motivates the maintainer to update the outdated requirements. We propose two options for displaying the outdated requirements. The first option is to present the impacted requirements in the form of a ranked list, where the requirements at the top are more likely to be impacted than the ones below. The second option is to display the complete requirements specification and use colour to highlight the parts that are likely to be impacted. The intensity of the colour used for highlighting a requirement reflects the likelihood of that requirement being impacted. Implementing one of these displaying approaches is a pure engineering problem. Therefore, it is not covered any further in the paper.

3.3 Identifying Relevant Changes

The code should implement the needs and requirements of the stakeholders. Therefore, the source code is changed whenever there is a need to adapt the behaviour of the system to new requirements. However, not all changes in source code affect the requirements. In fact, many of the code changes are refactoring, bug fixes, changes in implementation details, etc. So our goal

is to develop an approach that automatically identifies the code changes that are likely to affect requirements. First results of this approach have been published in [BCKG12]. These results include observations made during an empirical study about the relations between changes in the source code and changes in the external behaviour of the system. To make the paper self-contained, we include these observations in Section 3.3.1.

3.3.1 Exploratory Study of the Relations Between Changes in Code and Changes in Requirements

To find out which type of source code changes are likely to affect the requirements, we conducted an exploratory case study where we looked at relations between changes in source code and changes in requirements. We looked at some simple source code change patterns and their effect on requirements. Examples of changes we considered are changes in method bodies, changes in method signatures, addition/deletion of elements and changes in private elements.

As the focus of our work is mainly on functional requirements, we decided to consider a change as requirements-affecting if it affects the external behaviour of the system (the behaviour visible for the end user). Therefore, in the rest of this article we call all the changes that are likely to affect the external behaviour of a software system *relevant changes* and we call all the changes that

do not impact the external behaviour, such as refactoring and bug fixes, *irrelevant changes*.

We conducted the study on ZXing[1], an open source project for a barcode reader that is developed in Java.

The research question of this study is:

> RQ1: What heuristics can be used to identify source code changes that likely affect the external behaviour of the system?

We compared two versions of the source code and made observations about how to differentiate between changes that are likely to be related to changes in system behaviour and changes that are refactorings or bug fixes. In our exploratory study, we went through all the changes between the versions 1.6 and 1.7 of ZXing manually and studied how requirements-related changes differ from refactorings and bug fixes. Then we used some randomly selected packages, to study the changes in detail and confirm the observations we made. In this section, we present the six observations we made and what heuristics for identifying relevant changes we could derive from them.

Observation 1: Changes in method bodies are in most cases related to refactoring and/or bug fixes Changes in method bodies are among the most frequent changes that are

[1] http://code.google.com/p/zxing/

applied to the source code. However, they are not the most important ones in terms of affecting the external behaviour of the system. In fact, most of the changes observed in the bodies of methods in the explored project were minor changes that are either refactoring or bug fixes. We also noticed that the few changes in method bodies that relate to additions or extensions of features in the system came along with additions of new elements (e.g. new classes, methods or fields). For example, in the packages that we chose for a detailed exploration, we identified 33 changes in method bodies. 23 of these changes were due to refactorings and bug fixes (the majority, 19, being refactorings) and six changes were related to the additions of new features. For the other four changes we could not find what was the intent of the developer behind it. All the changes related to feature extension came along with additions of new elements in the code. Based on these observations, we derive the following heuristic: ignore the changes in method bodies.

Observation 2: Additions of new elements (classes; methods; package; fields) are usually related to the addition or extension of features We noticed that extension and addition of features are in most cases implemented through an addition of new elements in the code, where the names of these added elements usually reflect the implemented feature. It is important to note that there were some cases where the added element only relates to some implementation details. Therefore it is wrong to assume that *all* additions are extensions. However, we still can derive the following heuristic: additions of new elements (additions of

packages, classes, methods and/or fields) is likely to affect the external behaviour of the system.

Observation 3: Additions and removals of elements having similar names are usually rename operations When using normal differencing tools, renames are detected as an addition and a removal of two different elements. This can be very misleading as addition of elements is likely to relate to feature extension while renames are simple refactorings. When exploring the ZXing project, we noticed that in many cases, the new name is very similar to the old one (e.g. the field PDF417 was renamed to PDF_417). Therefore renames could be identified by computing the similarity between the name of the deleted and the name of the added one.

Observation 4: Changes in methods signature are usually related to refactoring Changes in methods signatures (other than renaming the method) were among the frequent changes that we observed when exploring the ZXing project and were in most cases related to refactoring. These changes can affect the visibility of the method (public, private, etc.), its return type (e.g. int, boolean, object...) and/or its parameters (e.g. the type of the parameters). In the packages we used for confirming the observation, all of the signature changes were due to refactoring. The heuristic we derive based on this observation is: ignore signature changes.

Observation 5: Changes in private elements can affect the external behaviour of the system When starting our exploratory study, we were expecting to find that changes in public elements are likely to affect the external behaviour of the system while private elements will only relate to implementation details. However, this was not the case in the ZXing project, as there were many changes in private elements that affected the external behaviour of the system. Therefore, we include changes in private elements when looking for changes affecting the system behaviour.

Observation 6: Additions of several methods having the same name are usually related to the same feature In many cases, we noticed that several methods having exactly the same name but different parameters were added to a class. In almost all cases, these methods related to the same feature and had similar behaviour. Therefore we derive the heuristic to consider and analyse only one of the added methods instead of considering them all.

3.3.2 Approach for Detecting Relevant Code Changes

In this section we present our approach for identifying the changes in the code that are likely to affect the external behaviour of the system and thus requirements. We built our approach based on the heuristics presented in Section 3.3.1, about the relations between

changes in code and changes in the system external behaviour. As the heuristics were based on observations made on a project written in an object-oriented programming language, the approach should work one similar project types. In the rest of this article, we assume that the source code on which we apply our approach is composed of the following elements: packages, classes, methods and fields.

Our algorithm for detecting relevant changes is composed of two parts: (1) the comparing part, where we compare all the elements in the code to detect the ones that have been added and removed and (2) the filtering part where we filter out the additions and removals that are due to renames.

In the comparing part, the main two changes our approach aims at detecting are (1) the addition and (2) the deletion of elements in the code, where an element can be a package, a class, a method or a field. When focusing on addition and removal only, we are likely to identify the changes related to feature extension, addition and deletion (Observation 2) while ignoring refactoring and bug fixes that show up as changes in method bodies (Observation 1) and in elements signature (Observation 4).

The comparison is done as follows. First, we compare the packages in both versions and detect those that have been added or removed. The comparison is based on the name only, therefore a package is considered as added to the new version (respectively removed from the old version), if there is no other package having the same name in the old version (respectively the new version). Second,

we go through each of the packages that exist in both versions, and for each package, we look for the classes that have been added or removed. Third, we go through each of the classes that exist in both versions and look for the methods and fields that have been added or removed. Both public and private elements are considered in the comparison as they both are likely to affect the external behaviour of the system (Observation 5).

As our comparison is done based on element names, renames are detected as a simultaneous addition and removal of two elements. Therefore, we try to identify renames and filter them out. For this, we consider the similarity between the names of the added and the removed elements as well as the call hierarchy of elements. If the added and deleted elements belong to the same parent element (e.g. two fields belong to the same class) and if they have similar names, then the change is considered as a rename (Observation 3) and is filtered out by our approach. There are several ways to compute the similarity between two strings of characters. One of the popular measures is the Levenshtein distance [Lev66], which is calculated as the minimum number of edits needed to transform one string into the other. The edits considered for the Levenshtein distance are insertion, deletion, or substitution of a single character. In the case of methods and classes, we also explore the call hierarchy of the elements: if the added and the deleted elements have the same call hierarchy, then there is a rename.

The output of this step is a set of source code changes that are composed of additions and deletion of elements and which are likely to affect the external behaviour of the system and thus its

requirements. These changes are then traced to the requirements specification as detailed in the next section. If no relevant changes are detected then the commit is considered not to be requirements-affecting and is thus ignored.

3.4 Tracing Changes to the Requirements

In this section, we discuss the tracing strategy for identifying outdated requirements. The tracing approach is composed of two steps. In the first step, we prepare the data to be traced to the requirements (see Section 3.4.1). This data is composed of keywords extracted from the code to describe the change. The keywords are grouped by class. In the second step, we trace the extracted keywords to the requirements using a tracing approach that is based on Information Retrieval (IR) and we generate a ranked list of the requirements that are likely to be impacted (see Section 3.4.2).

3.4.1 Keywords Extraction

The goal of this step is to extract as much relevant information as possible about the change in order to trace it to the requirements. The tracing is done based on the textual similarity between the information describing the change and the requirements. Therefore, it is important to gather as many relevant keywords as possible

about the change and its context, but at the same time the keywords should also be specific enough to the change in order to increase the precision of the tracing. In our approach for extracting keywords we consider three sources of information: (1) the name of the elements affected by the change, (2) the call hierarchy of the changed elements and (3) their documentation. We detail how we extract the keywords from each of these sources in the remainder of this section.

Using meaningful names when coding is one of the most important coding practices. Therefore, in many projects, the name of an element reflects its intended behaviour (for example in a library management system the method for managing book borrows is likely to be called *borrowBook*). We consider names of the changed (added or removed) elements as a valuable source of information about the change itself. If the changed element is a class, we will consider its name and the names of the methods and fields it contains. If the changed element is a method or a field, then we consider its name as well as the name of its parent class. The reason is that parent and sub-elements usually give information about the context of the change. As the elements names are likely to contain several keywords, we split these names into keywords based on the naming convention used (e.g. the camelCase convention). Then we include these keywords in the list of terms to be traced to the requirements. Column 2 of Table 3.1 presents the elements names that we include in the list for each type of changed element.

The documentation of classes and methods are likely to include a description of the behaviour and purpose of the element in natural

Table 3.1: Elements used for extracting keywords for each type of change

Changed Element	Names	Documentation	Call Hier.
Package	package, sub-classes	Yes	No
Class	class, sub-methods, sub-fields	class	Yes
Method	method, parent class	method, parent class	Yes
Field	field, parent class	parent class	No

language. Therefore we also include the keywords contained in the documentation of change-related elements to the list of terms to trace (see Column 3 of Table 3.1).

To get the context of the change, we do not only consider the changed elements, but also their call hierarchy. By call hierarchy we mean all the methods/classes that are invoking the changed method or those invoking the constructor of the changed class. The invocation can be either direct or via other methods/classes. Details about when the call hierarchy is used are given in column 4 of Table 3.1. Elements from external packages are not in the call hierarchy and the depth of the call hierarchy can be set by the user.

We group the terms related to the change by class. The reason behind this is that considering each single change separately is very fine grained, as a change in a single element might not be relevant by itself. On the other hand, elements contained in one class usually relate to the same concept, therefore, grouping changes by classes results in more keywords relating to the change without loosing the specificity of these keywords to the change. Finally, we

filter out stop words and common words in the project in order to reduce the number of irrelevant terms in the list.

3.4.2 Tracing

After doing the keyword extraction, we obtain several lists of keywords, where each list contains the terms related to the changes in one class. We trace these keywords to the requirements specification using an information retrieval (IR) based tracing technique. The advantage of the IR-based tracing is that it is fully automated. There are several IR-based tracing techniques and tools available and which can be used, such as Retro [HDS+07], which we used for the evaluation. The results of the IR-based tracing is a ranked list of requirements for each changed class. As there are several changed classes, we get a separate ranked list for each class. We use these lists to compute a final list that indicates the relevant requirements to the maintainer. We compute the final list in the following way: We give a score to each requirement appearing in the lists generated by the IR-based tool, according to their rank. Let us assume that we are tracing to a requirements specification that includes 300 requirements. Then, the first requirement appearing in each of the ranked lists will get the score 300, then the second requirement will have the score 299, and so on. Afterwards we sum, for each of the requirements, all the scores from the different lists to get the final score of the requirement. Then we sort the requirements according to their final scores. With this method, the rank of a requirement depends both on its rank in

the initial lists and on how often it appears in the lists. This will allow filtering out the ranks obtained from tracing a changed class that is either irrelevant or very generic.

3.5 Tools

Applying our approach is meaningful only when it is tool-supported. Therefore, we developed a prototype that automatically runs the different steps of the approach. The prototype is composed of three tools which are presented in Figure 3.2. The *Comparator* detects the relevant changes in the code and extract keywords describing the changes. *Retro* traces the keywords to the requirements and generates a ranked list of impacted requirements for each class. The *Tracing macro* combines the results obtained from Retro and generates the final list of impacted requirements.

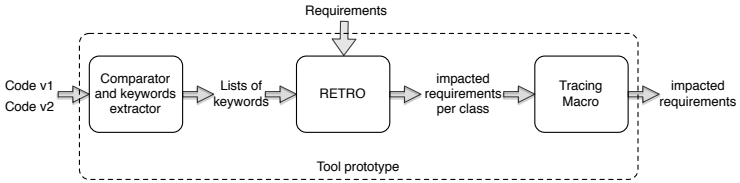

Figure 3.2: Prototype tool set implementing our approach

Comparator The Comparator tool compares two versions of source code, detects relevant changes and extracts keywords describing each of the changes. The tool is based on an existing

Java library to compare Java API called JDiff [Doa02]. JDiff has several similarities with our comparing technique, as it detects addition and removal of elements in the code and ignores changes in method bodies. We adapted JDiff to our approach by making it ignore changes that are not relevant in our case, such as changes in methods signature. We configured the tool so that it detects changes in private elements. We also implemented a comparator that compares the names of added and removed elements and their call hierarchy in order to detect renames. Name comparing is based on the Levenshtein distance. For the experiments we run with the tool, renames are detected if they have a Levenshtein distance that is equal or less than 2 or if they have the same call hierarchy and a Levenshtein distance that is equal or less than 5. The tool automatically extracts keywords from changed classes as presented in Section 3.4.1, and generates a list of textual files, where each file contains the keywords related to relevant code changes in one class. In the current implementation of the tool, extracting keywords from package documentation is not supported yet. However, this has no effects on the results of the evaluation we made, as in both projects we used there was no addition or removal of packages. For the experiment, we set the call hierarchy depth to 2.

Retro For the tracing we used an existing tool called REquirements TRacing On target (Retro) [HDS+07]. Retro, which is an IR-based tool, takes as input two lists of textual documents and returns a list of candidate links that are ranked based on the similarity between the documents. In our case, we give Retro the list

of files, where each file contains the keywords related to a changed class, which we obtained from the comparator, and a list of files containing the requirements (one requirement per file) as input. We used Retro with the default configuration, which is the *vector space retrieval* with *tf-idf (term frequency - inverse document frequency)* term weighting. Retro includes other functionalities, which we did not use in our evaluation, such as entering analyst feedback to improve the tracing. As output, we obtain a list of requirements for each changed class. The requirements are ranked according to their similarity to the keywords Retro also assigns values representing the similarity, but we do not use these values in the current version of our approach.

Tracing Macro To obtain the final list of impacted requirements, we merge the lists obtained from Retro using a macro that implements the scoring technique presented in Section 3.4.2. The output of the macro is one list of requirements that are ranked according to their likelihood of being impacted by the change. The comparator tool, the Retro tool and the macro can be easily integrated into one tool if needed.

3.6 Evaluation

3.6.1 Goal and Metrics

To evaluate our approach, we applied it to two case studies: AquaLush, a software project for managing an irrigation system

and iTrust, a tool for managing medical data. For our evaluation we use the Goal-Question-Metric method. We define our goal according to the template developed by Basili in [Bas92]. The goal of the evaluation is to:

> Analyse *our approach for identifying outdated requirements* for the purpose of *evaluation* with respect to *the correctness of the output* from the point of view of *maintainers* in the context of *the co-evolution of textual requirements specification and object-oriented source code*

To assess the goal, we define the following two questions.

Q1: How good is the approach in identifying the commits that affect requirements and in filtering out those that do not affect requirements?

This question relates to the first part of our approach, where we detect whether a commit is likely to affect requirements or not based on the heuristics presented in Section 3.3.1. Although the final goal of the approach is to identify the outdated requirements, identifying whether a commit affects requirements or not is also useful information for the maintainer in the context of requirements update. In fact, it informs the maintainer whether the new code changes are likely to introduce inconsistencies with the current requirements specification or not. Successfully filtering out the commits that do not impact requirements is also important as it reduces the number of irrelevant interruptions for maintainers and thus increases their trust in the approach.

Q2: For relevant code changes, how good is the approach in identifying the impacted (outdated) requirements?

Our approach allows identifying the outdated requirements statements among the different statements in the requirements specification so that the maintainer does not need to look for them manually. In this question, we evaluate how good is the approach in identifying as many outdated requirements as possible without introducing many requirements that are not impacted by the change.

Metrics for Question 1 As Q1 is about classifying whether a commit is requirements-affecting or not, we use three measures from the classification context, which are the *true positive rate* (also called *sensitivity* and called *recall* in information retrieval context), the *true negative rate* (also called *specificity*) and the *overall accuracy* ([OD08] page 138) . The true positive rate (TPR) evaluates how good is the approach in identifying as many relevant commits as possible and the true negative rate (TNR) evaluates how good is the approach in ignoring as many irrelevant commits as possible. The overall accuracy (A) evaluates the ability of the approach to identify as many relevant commits as possible and to ignore as many irrelevant commits as possible.

These measures are calculated based on a coincidence matrix (see Table 3.2) that includes the number of relevant commits that are detected (true positives = TP), the number of irrelevant commits that are ignored (true negatives = TN), the number of irrelevant

Table 3.2: A simple coincidence matrix [OD08]

		True Class	
		Positive	Negative
Predicted Class	Positive	True Positive Count (TP)	False Positive Count (FP)
	Negative	False Negative Count (FN)	True Negative Count (TN)

commits that are detected (false positives = FP) and the number of relevant commits that are ignored (false negatives = FN). More formally the measures are defined as follows:

M1: Accuracy (A) is obtained by dividing the total correctly classified positives and negatives by the total number of samples:

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

M2: True positive rate (TPR) is obtained by dividing the correctly classified positives by the total positive count:

$$TPR = \frac{TP}{TP + FN}$$

M3: True negative rate (TNR) is obtained by dividing the correctly classified negatives by the total negatively classified count:

$$TNR = \frac{TN}{TN + FP}$$

In the ideal case, where the approach returns true positives and true negatives only, we get $A = 1$, $TPR = 1$ and $TNR = 1$ and in the worst case, where only false negatives and false positives are returned, we get $A = 0$, $TPR = 0$ and $TNR = 0$. Therefore, we aim at obtaining values of A, TPR and TNR that are as high as possible.

Metrics for Question 2 As question Q2 is about identifying a certain class of requirements (the outdated ones) from a large set of requirements, we use metrics from the information retrieval field to assess how good the results are. The traditional measures that are used to evaluate the ability of a retrieval method to return relevant answers are *recall* R, *precision* P and $fallout$ F [Dom08]. The recall (which is similar to the TPR for Q1) evaluates how good is the approach in identifying as many outdated requirements as possible. The precision evaluates how good is the approach in identifying outdated requirements only. The fallout (which is equivalent to 1-TNR for Q1) evaluates how many irrelevant requirements (not outdated) are wrongly detected among the total number of irrelevant requirements. In the ideal case, where the approach returns the outdated requirements only, we get $R = 1$, $P = 1$ and $F = 0$ and in the worst case, where only requirements that are not outdated are returned, we get $R = 0$, $P = 0$ and

$F = 1$. Therefore, we aim at obtaining values of R, and P that are high with a value of F that is as low as possible.

As it is easier for the maintainer to find the outdated requirements when they are suggested early in the ranking, we use recall R_n, precision P_n, and fallout F_n at cut-off rank n, i.e. we only consider the requirements identified with a rank that is lower than n (cf. [Dom08, Sec. 4.9.3]).

To define the new metrics, we use the same terminology used in the classification context (TP, TN, FP, FN). The reason is that information retrieval is also a kind of classification, as the elements are classified as relevant (which should be retrieved) and irrelevant (which should be ignored). The coincidence matrix for Q2 includes, for each commit i, the set of outdated requirements that are detected with a rank lower than n by the approach ($\mathrm{TP_{in}}$), the set of requirements that are not outdated and are detected with a rank lower than n ($\mathrm{FP_{in}}$), the set of requirements that are not outdated and are not detected with a rank lower than n ($\mathrm{TN_{in}}$), and the set of requirements that outdated and are detected not detected with a rank lower than n ($\mathrm{FN_{in}}$).

More formally, the measures are defined as follows:

M4: R_n is the average recall for all commits:

$$R_n = \frac{\sum_{i=1}^{k} r_{in}}{k}$$

Where k is the number of commits considered and r_{in} (the recall for commit i) is the proportion of identified documents out of the outdated ones at a cut-off rank n:

$$r_{in} = \frac{TP_{in}}{TP_{in} + FN_{in}}$$

M5: P_n the average precision for all commits:

$$P_n = \frac{\sum_{i=1}^{k} p_{in}}{k}$$

Where k is the number of commits considered and p_{in} (the precision for commit i) is the proportion of outdated requirements out of those identified at a cut-off rank n:

$$p_{in} = \frac{TP_{in}}{TP_{in} + FP_{in}}$$

M6: F_n the average fallout for all commits:

$$F_n = \frac{\sum_{i=1}^{k} f_{in}}{k}$$

Where k is the number of commits considered and f_{in} (the fallout for commit i) is the proportion of identified requirements out of the ones that are not outdated at a cut-off rank n:

$$f_{in} = \frac{FP_{in}}{TN_{in} + FPin}$$

To study how early relevant links are suggested by the approach, we increase the cut-off rank n which starts from 1 and create a precision-recall graph as well as a fallout-recall graph.

3.6.2 Case studies and Experimental Setup

To do the evaluation, we used two case studies: AquaLush and iTrust. For each of the case studies, we manually identified the ground truth (the relevant commits and the outdated requirements for each commit). Then we run the tool presented in Section 3.5 and we evaluated the metrics we defined previously. In this subsection, we present each of the case studies we used, as well as how we built the ground truth for it.

Case Study 1: AquaLush

Description AquaLush is a software project for managing an irrigation system that has been originally developed as an illustrative example for a book about software design [Fox06] and has later been extended and used as a benchmark for traceability [BCCJG11]. AquaLush has a structured requirements specification that is written in natural language. The source code of AquaLush is written in Java with around 11KLOC. The requirements specification of AquaLush, its source code, as well as all other AquaLush artifacts can be found at the webpage of the AquaLush benchmark[2].

To perform the tracing part of the experiment, we had to delete the section titles as well as the figures in the requirements specification, and we included each requirement statement in a separate textual file. This is because Retro only accepts a flat list of textual files as

[2]http://www.ifi.uzh.ch/rerg/research/aqualush.html

Table 3.3: List of changes applied to AquaLush

Change 1	Allow setting the water allocation for each of the zones separately
Change 2	Add a maximum moisture level: the irrigation should start when the moisture level is lower than the critical moisture level and should stop as soon as the maximal moisture level is reached. The default max level should be 50.
Change 3	Create a log that includes the timestamp for each of the following events: change irrigation mode, setting water allocation, setting irrigation time, setting critical or maximum moisture level. A button *Show log* allows the user the access the log. There should be two buttons that allow the user to browse the log up and down.
Change 4 (bug)	When setting the *Times real time* to the maximum (1000), there is a problem in the screen for *control irrigation* (the screen is blinking)
Change 5 (bug)	The SimStorageDevice is not working as it does not store any data.
Change 6 (bug)	In class *Zone* , the method *setIsFailed*, does not set the valve to *closed* after setting it to *failed*. This might lead to the problem that a device is set as failed and open at the same time.
Change 7 (bug)	If there is already less than one hour before the next irrigation time, the *jump* button should has no effect. However in the current implementation, jump sets the time to the next irrigation day.
Change 8 (bug)	It is impossible to set water allocation to 0: when setting the water allocation to 0 and clicking on *Accept new settings* then going back to set water allocation we find that the old value of allocation is restored.

input for the tracing. The total number of statement considered in the experiment is 337.

As there is only one release of AquaLush available, we had to develop a second release to run our experiment. Therefore, we prepared a list of eight changes and we asked a developer to implement these changes. The changes include three new features and five bug fixes. The list of changes is presented in Table 3.3.

In order to limit the threats to validity when developing the new version, we asked an external developer to implement the changes. The developer is not involved in our work, and does not know why we are developing the new AquaLush release. In order to avoid the risk of having all the changes in one commit, we asked the developer to commit the code after implementing each of the changes and to mention in the commit message what change has been implemented. The developer did an additional commit to fix a bug that was introduced when implementing one of the features. Therefore we had 9 commits in total.

Ground Truth The ground truth is composed of two components, which are (1) the set of commits that impact requirements and (2) the set of outdated requirements for each requirements-impacting commit. Deciding which commits are requirements-affecting is straightforward. In fact, we know already which of the implemented changes affect requirements and we know what change is implemented by each commit. In total we have six commits that do not impact requirements (commits for changes 4, 6, 7, 8 and the additional bug fix made by the developer) and 3 commits that impact requirements (commits for changes 1, 2 and 3). To identify outdated requirements, we went manually through the requirements specification of AquaLush and identified what requirements are impacted by change 1, 2 and 3. We identified 8 outdated requirements for change 1, 6 outdated requirements for change 2 and three outdated requirements for change 3.

Other than the outdated requirements, we noticed that there are also many requirements that are directly related to the change

although they are not outdated in the sense that they are still consistent with the new implementation. We also marked the related requirements as we think that they can give information about the context of the change. We identified 8 related requirements for change 1, 24 for change 2 and 4 for change 3. According to the metrics we presented previously, our approach is considered successful if it detects the outdated requirements. Detecting the related requirements can be interesting as well as it might support the maintainer, but it is not compulsory, therefore we do not consider it in our metrics.

Case Study 2: iTrust

Description The second case study we used for the evaluation is the iTrust Medical care project [MSW12], which is a tool for managing medical data that is developed using a combination of Java code and Java Server Pages. The tool was developed for teaching purposes at the North Carolina University and it has several code releases and a wiki-based requirements specification which includes functional requirements, non-functional requirements, a glossary, a set of global constrains and a section for specifying the data format for the input fields. Every semester, students apply new changes to iTrust, and a new source code version is released at the end of the semester. In our experiment, we used the functional requirements of iTrust, which are specified in the form of fine-grained uses cases (See Figure 3.3). There are 39 use cases in total.

For the code, we only considered the part written in Java as our prototype only works on Java code. We used versions 10 (release

UC1 Create and Disable Patients Use Case

1.1 Preconditions: The iTrust HCP has authenticated himself or herself in the iTrust Medical Records system [UC3].

1.2 Main Flow: An HCP creates patients [S1] and disables patients [S2]. The create/disable patients and HCP transaction is logged [UC5].

1.3 Sub-flows:

[S1] The HCP enters a patient as a new user of iTrust Medical Records system. Only the name and email are is provided. An email with the patient's assigned MID and a secret key (the initial password) is personally provided to the user, with which the user can reset his/her password. The HCP can edit the patient according to data format 6.4 [E1] with all initial values (except patient MID) (...)

[S2] The HCP provides the MID of a patient for whom he/she wants to disable [E2]. The HCP provides a deceased date (data format 6.4). An optional diagnosis code is entered as the cause of death.

1.4 Alternative Flows:

[E1] The system prompts the enterer/editor to correct the format of a required data field because the input of that data field does not match that specified in data format 6.4 for patients.

[E2] (...)

Figure 3.3: Example use case from iTrust requirements specification, version of September 3rd, 2010 [WXM+]

date: August 18th, 2010) and 11 (release date: January 7th, 2011) of the source code. To obtain the requirements that correspond as much as possible to each of these releases, we choose a wiki version from a date that is after the code release and before the beginning of the following semester. The reason is that, after the release, the project owners do a cleanup and maintenance for the requirements based on the work done by the students. Therefore we consider the requirements specification as of September 3rd, 2010 (the "old requirements") for the source code version 10 (the "old code version") and the requirements as of February 7th, 2011 (the "new requirements") for the source code version 11 (the "new code version") [WXM+].

Ground Truth The challenge we had in building the ground truth for iTrust was that the commits were not available. Therefore, we had to manually compare the two version of source code and classify all the changes according to whether they affect requirements or not. Then we grouped the changes that relate to the same conceptual change, i.e. the same feature, and considered them as one commit. We found 14 conceptual changes, which are presented in the second column of Table 3.4.

For the changes that did not impact requirements, we could not group them by commit as it was impossible to find how they were distributed. Consequently, we could not estimate the count of false positive (FP) and true negative (TN), which are needed to calculate the accuracy A and true negative rate TNR for Q1. Therefore, we added new metrics which are $\tilde{A}$, $T\tilde{P}R$ and $T\tilde{N}R$ for

Table 3.4: Identified changes and affected use cases

Conceptual change	Change description	Affected use cases
Change 1	Activity feed	none (new req.)
Change 2	Enabling appointment editing	UC22
Change 3	Uploading photo	UC4
Change 4	Reason code	UC15, UC37
Change 5	Weight/height charting	UC10
Change 6	Login (added captcha and attempts)	UC3
Change 7	Office visit form (added orc and comment)	UC11
Change 8	Remote monitoring (added height, weight, etc.)	UC34
Change 9	Remote monitoring (get patient data by type)	UC34
Change 10	Display patient's monitoring HCP	UC34
Change 11	Cause of death validation	UC1
Change 12	Notes (format change)	UC11
Change 13	Logging (added logs)	UC5
Change 14	Colour options	none (new req.)

detecting code classes that include requirements-affecting changes. The formulas for $\tilde{A}$, $T\tilde{P}R$ and $T\tilde{N}R$ are similar to those for A, TPR and TNR, with the difference that

- TP is the number of detected classes that contain relevant changes,
- TN is the number of classes containing irrelevant changes which are ignored,
- FP is the number of classes containing irrelevant changes that are detected and
- FN is the number of classes containing relevant changes and which are ignored.

There were 91 classes that changed in total, among which we found 31 that contain requirements-related changes.

The change we made for evaluating Q1 does not affect the metrics used for Q2. To identify the outdated requirements for each of the conceptual changes, we used the same ground truth that we built for a previous experiment in [BCKG12], and where we went manually through the use cases and identified the ones that are outdated. Although the current experiment is different from the one performed in [BCKG12], as now we trace all the classes related to a conceptual change together, the ground truth is the same for both experiments. The outdated requirements for each of the conceptual changes are reported in the third column of Table 3.4.

3.6.3 Results

In this section, we report the results obtained for each of the case studies.

AquaLush

Detecting Relevant Commits (Q1) When checking for relevant commits, our approach detected four out of the nine AquaLush commits as relevant. The result for individual commits is reported in Table 3.5. Three of the commits were actually relevant. There are in total 3 true positive, 5 true negative, 1 false positive and 0 false negative. The accuracy (M1), true positive rate (M2) and true negative rate (M3) we obtained are : $A = (3 + 5)/(3 + 5 + 1 + 0) = 88.8\%$, $TPR = 3/(3 + 0) = 100\%$ and $TNR = 5/(5 + 1) = 83.3\%$

Table 3.5: List of changes applied to AquaLush

Change Commit	Relevant	Correctly detected
Change 1	Yes	Yes
Change 2	Yes	Yes
Change 3	Yes	Yes
Change 4	No	Yes
Change 5	No	No
Change 6	No	Yes
Change 7	No	Yes
Change 8	No	Yes
Additional commit	No	Yes

Detecting Outdated Requirements (Q2) To answer Q2, we evaluated the recall (M4), precision (M5) and fallout (M6) for various cut ranks n. In Table 3.6, we report the list of requirements obtained for a cut rank $n = 25$, and we colour the outdated requirements as well as the related ones. As mentioned earlier, the related requirements are not outdated as they are consistent with the new implementation although they are directly related to the change. We highlighted them as we think that they can give the user information about the context of the change but they are not considered in the metrics we are using. Each requirements has an identifier which is composed of the term SRS and then the ID number of the requirement as found in the benchmark for traceability. We report the precision/recall values and fallout/recall values obtained for various cut ranks in Figure 3.4 and Figure 3.5 respectively.

The Maximum precision value we obtained is $P_{20} = 23.3\%$ for a recall of $R_{20} = 73.6\%$ and a fallout of $F_{20} = 4.6\%$. This means that by looking at 20 requirements only after each commit, the

Table 3.6: Identified requirements for AquaLush changes with a ranking below 25

Rank	Change 1	Change 2	Change 3
1	SRS383	SRS358	SRS237
2	SRS33	SRS24	SRS24
3	SRS384	SRS367	SRS97
4	SRS254	SRS28	SRS53
5	SRS53	SRS366	SRS293
6	SRS30	SRS253	SRS69
7	SRS378	SRS368	SRS252
8	SRS31	SRS369	SRS238
9	SRS52	SRS377	SRS277
10	SRS32	SRS27	SRS52
11	SRS379	SRS97	SRS98
12	SRS145	SRS376	SRS291
13	SRS29	SRS25	SRS48
14	SRS186	SRS360	SRS240
15	SRS24	SRS187	SRS112
16	SRS48	SRS26	SRS32
17	SRS153	SRS374	SRS109
18	SRS69	SRS359	SRS333
19	SRS380	SRS53	SRS110
20	SRS98	SRS69	SRS150
21	SRS47	SRS52	SRS289
22	SRS97	SRS113	SRS27
23	SRS382	SRS167	SRS99
24	SRS49	SRS277	SRS96
25	SRS381	SRS109	SRS269
Missed	0	0	2

Outdated
Related

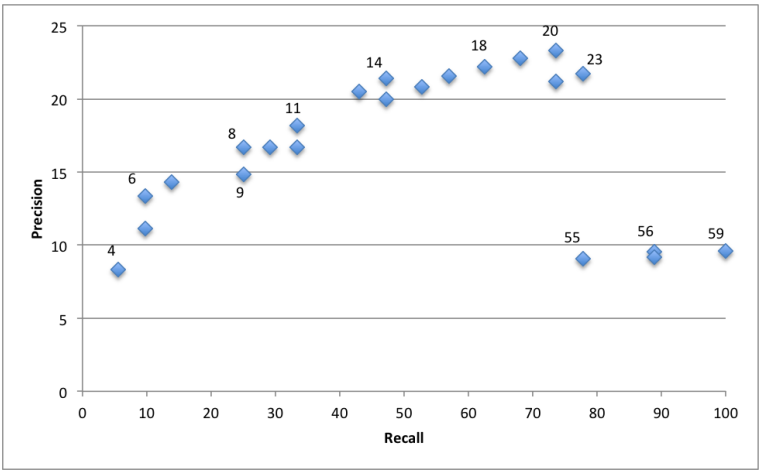

Figure 3.4: AquaLush: precision / recall at different cut ranks, the respective cut rank n is annotated to the data points

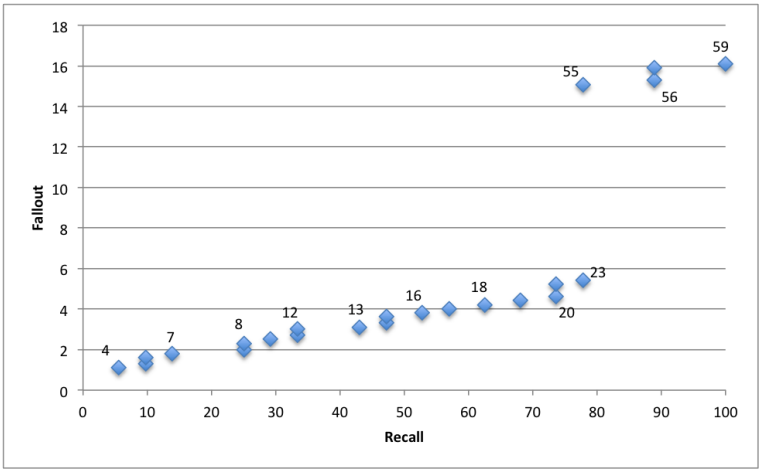

Figure 3.5: AquaLush: fallout / recall at different cut ranks, the respective cut rank n is annotated to the data points

maintainer is able to detect 73% of the outdated requirements, and that only 4.6% of the requirements that are not outdated will be suggested to the maintainer as outdated, while the rest, 95.4% of the requirements that are not outdated, will be filtered out.

Full recall, $R_{59} = 100\%$ is obtained at cut rank $n = 59$ for a precision of $P_{59} = 9.6\%$ and a fallout of $F_{59} = 16\%$. This means that the list proposed to the maintainer includes all of the outdated requirements and filters out 84% of those that are not outdated.

iTrust

Detecting Relevant Commits (Q1) For the iTrust project, 12 out of the 14 conceptual changes were detected as relevant. So we get 12 true positive and 2 false negative. This gives a recall of $TPR = 12/(12 + 2) = 85.7\%$. When considering classes instead of commits (see Section 3.6.2), the approach detected 33 classes among which 26 actually contained requirements-related changes, and it ignored 53 out of the 60 classes containing irrelevant changes. So we get 26 true positives, 7 false positives, 53 true negatives and 5 false negatives. This results in $\tilde{A} = (26+53)/(26+53+7+5) = 86.8\%$, $T\tilde{P}R = 26/(26+5) = 83.8\%$ and $T\tilde{N}R = 53/(53+7) = 88.3\%$

We didn't obtain a recall of 100% because two conceptual changes were missed. The two missed changes in iTrust are the following. (1) A change of the input format for notes, where a hash tag has

been added as an allowed character. As this change only was made inside a String constant in the `ValidationFormat` enumeration class, our tool did not detect it. (2) An addition of a new condition in the `PatientValidator` class to validate that no patient can be marked as dead unless the cause of death is specified. As this was implemented by adding an "if" statement inside the body of a method, it could not be detected by our tool. To summarize, both undetected changes were related to the validation of input forms only.

Detecting Outdated Requirements (Q2) In this paragraph we report the recall, precision and fallout obtained when tracing the conceptual changes that were detected in the previous step to the requirements specification of iTrust. It is important to mention that we did not consider the changes that only result in addition of new requirements without implying any changes in the existing requirements (changes 1 and 14) because our current approach addresses outdated requirements and not missing ones.

The precision/recall values and fallout/recall values we obtained for different rank cuts are presented in Figure 3.6 and Figure 3.7 respectively. We also report the ranks of the outdated use cases as identified by our approach in Column 3 of Table 3.7.

The Maximum precision value we obtained is at cut rank 1, where we have $P_1 = 44.4\%$, $R_1 = 44.4\%$ and a fallout of $F_1 = 1.4\%$. Full recall, $R_{34} = 100\%$ is obtained at cut rank $n = 34$ for a precision of $P_{34} = 3.2\%$ and a fallout of $F_{34} = 86.8\%$. At rank cut $n = 7$,

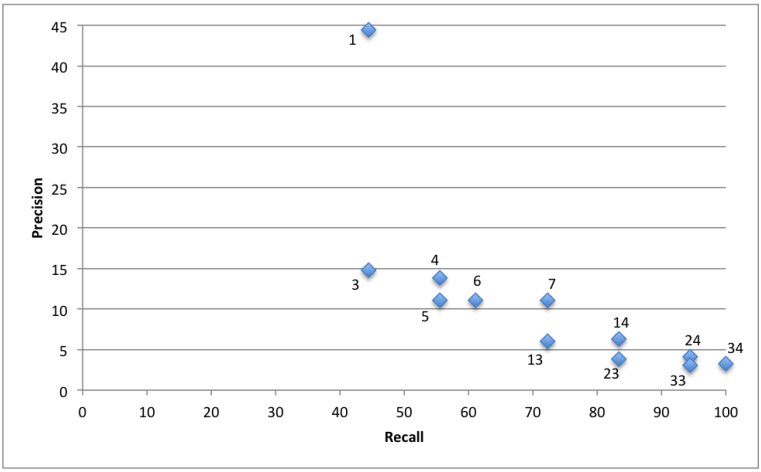

Figure 3.6: iTrust: precision / recall at different cut ranks, the respective cut rank n is annotated to the data points

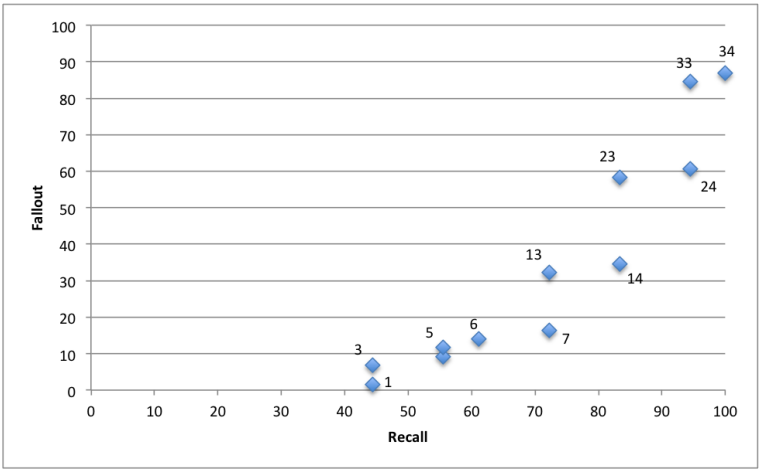

Figure 3.7: iTrust: fallout / recall at different cut ranks, the respective cut rank n is annotated to the data points

Table 3.7: iTrust rank results

Change	affected use cases	Rank of outdated use case
Change 1	None	New requirement
Change 2	UC22	14
Change 3	UC4	24
Change 4	UC15, UC37	6 | 34
Change 5	UC10	7
Change 6	UC3	1
Change 7	UC11	4
Change 8	UC34	1
Change 9 and 10	UC34	1
Change 11	UC1	Change not detected
Change 12	UC11	Change not detected
Change 13	UC5	1
Change 14	None	New requirement

the recall is $R_7 = 72.2\%$ for a precision of $P_7 = 11.1\%$ and a fallout of $F_7 = 16.4\%$. This means that the maintainer can detect more than 70% of the outdated requirements by looking at a list of 7 requirements after each commit, and that the list includes only 16.4% of the requirements that are not outdated.

When looking carefully at the ranks reported in Table 3.7, we notice that most of the ranks are good (7 or better) except for changes 2, 3 and 4. When investigating the reasons behind the bad ranks we found the following. For change 2, the problem was related to the use of abbreviated terms in the code, namely the term *appt* was used as abbreviation to *appointment*. The abbreviated terms could not be matched to the complete terms in the requirements specification and this resulted in the bad rank. Changes 3 and 4 were extensions of existing features, so new names terms have been added which did not appear in the requirements specification. Additionally, for both of these cases, the new elements that have

been added to the code were called from the jsp classes only, so there was no call hierarchy available and thus we could not gather more information about the context of the change. Therefore, the tracing of these changes to the code was not efficient.

3.6.4 Summary

The results obtained from the experiment are summarised in Table 3.8. For both case studies, our approach succeeded to detect the relevant commits with an accuracy higher than 85%, a sensitivity (TPR) higher than 80% and a specificity (TNR) that is higher than 80%. For the identification of outdated requirements, a recall higher than 70% could be obtained for both studies while filtering out more than 80% of the requirements that are not outdated.

3.7 Discussion

3.7.1 Approach Evaluation

Based on the results obtained in the evaluation, we can claim that our approach succeeded to achieve its goals of identifying relevant code commits and identifying the outdated requirements relating to them. In fact, for both projects, the approach detected most or all of the relevant commits and most or all of the requirements related

Table 3.8: Results summary

		AquaLush		iTrust	
Detecting relevant commits (Q1)	**Accuracy**	88.8		86.8	
	TPR (sensitivity)	100		83.8	
	TNR (specificity)	83.3		88.3	
Detecting outdated requirements (Q2)	*cut Rank*	*n=23*	*n=59*	*n=1*	*n=7*
	Recall	77.7	100	44.4	72
	Precision	21.7	9.6	44.4	9.7
	Fallout	5.4	16	1.4	19

to them while filtering out more than 80% of the irrelevant commits and more than 80% of the irrelevant requirements. Filtering out most of the irrelevant requirements should reduce the effort the maintainer needs to update the specification. Additionally, the approach is fully automated, so no additional effort is required for using it. As our approach gave good results for two case studies which have different characteristics (system type, type and structure of requirements specification, number of requirements, etc.), we expect it to perform well for other software systems as well.

The main two limitations of our approach are that (1) it might ignore some of the relevant commits and (2) it might miss some of the impacted requirements. The first problem relates to the

compromise between identifying all relevant changes and identifying relevant changes only. If we extend the approach to cover more code changes patterns, then the precision of the approach will decrease and there will be more irrelevant commits that are considered. The second problem is due to the limitation of the IR-based tracing techniques. In the IR-based tracing, term similarity is used to identify related documents. Therefore, these approaches are not able to identify documents that are conceptually related but which do not include similar terms. On the other hand, the main advantage of IR-based approaches is that they are fully automated.

Despite these limitations, our approach can still be very useful to maintainers. In fact, the goal of our work is not to replace maintainers but to support them in the update of the requirements. Allowing maintainers to rapidly find the impacted requirements for most of the changes is expected to encourage them to keep the specification up-to-date. Maintainers should however keep in mind that the approach can also miss some of the outdated requirements in some cases. In such cases, if the maintainer is familiar with the specification, then he might spot that something is missing and thus try to find it by doing an additional search. If the maintainer is not able to spot that an outdated requirement is missed, then this can include some inconsistencies in the requirements specification. The inconsistency problem can however be worse if the maintainer has to detect the outdated requirements fully manually in a long requirements specification. In fact, such manual tasks require a lot of effort and are error-prone. So it is very likely that the maintainer misses many of the outdated requirements, or even not update the

specification at all. Therefore, we expect our approach to reduce the inconsistency problem although the generated results are not 100% correct.

3.7.2 Scope

Our approach can be used when an initial requirements specification exists but is not kept up-to-date due to time and cost constrains. For maintainers who think that updating requirements is only a waste of time and does not bring any benefits, our approach is useless. In fact, our approach is only meant to reduce the effort required to update requirements but it does not eliminate the required effort completely nor does it force the maintainer to update the requirements.

The current differencing algorithm of the approach is designed for code written in object-oriented programming language. Extensions to other kinds of programming languages are possible but they are not in the scope of this work. The performance of the tracing part of the approach depends on the comments existing in the code. Therefore our approach is expected to work much better for code that is well commented than for uncommented or badly commented code.

Our approach detects the requirements that are impacted by the code changes, but it does not detect missing requirements in the specification. However, it can still support the maintainer in adding new requirements by finding the existing requirements that are related to the new one, and thus help the maintainer decide how and where to add the new requirement in the specification.

3.8 Related Work

Managing the evolution of requirements is a problem that has been addressed by several researchers from the software engineering field. Hermann et al. [HWP09] address the problem of specifying new requirements by proposing an approach to specify delta requirements in detail while describing the rest of the system on a higher-level of granularity. Zowgi and Gervasi [ZG03] explore ways to ensure that the requirements specification is correct, consistent and complete after each change using different validation checks. However, we are the first, to the best of our knowledge, to propose an automated approach for identifying the outdated requirements of a software system based on the changes applied to code. What is also special in our work is that we assume that the code is changed before the requirements while for most existing approaches for managing the evolution and update of requirements, the authors assume that the changes are applied at the requirements level first and then are propagated to the lower level artifacts and source code, as in [EBM11].

Our work relates to co-evolution as it aims at supporting the co-evolution of the code and the requirements specification. Most of the existing co-evolution approaches address the co-evolution of the implementation with the design [MKPW06] [CPGS07] [DVMW02]. Our work is more challenging for two reasons. First, we have to analyse the a document that is written in natural language (the requirements). Second, unlike design and implementation, which both relate to the solution domain, the requirements relate

to the problem domain, which makes the mapping between the requirements and the code more complex.

Software traceability is also one of the main approaches that are meant to support the maintenance of software artifacts. There are several approaches for automatically generating traceability links between software artifacts as well as approaches for using the traceability links to manage software change and evolution [HDS06] [ACC+02] [CHCC03] [CHCGE10]. Traceability can be very useful for propagating changes between artifacts. However, it has two main limitations. First, the use of traceability is still very limited in practice because of the high costs relating to the defining and maintaining of traceability links. Second, there is usually much scattering and tangling between requirements and code, which results in a high number of links that can be overwhelming for the maintainer.

In [BCG10], we propose an approach that uses the changes in high-order tests (such as acceptance tests) to identify outdated requirements. For this, a set of traceability links between the requirements and the acceptance tests are required. Using these links, we can trace back all modified tests to the requirements they derive from. The advantage of our current approach is that no tests or traceability links are required as the analysis is done on the source code directly and is automatically traced back to requirements.

3.9 Conclusion and Future Work

In this article, we presented an approach for identifying outdated requirements based on source code changes. Our approach is meant to support maintainers in the update of the requirements specification by automatically identifying the parts that are likely to be outdated after each code commit. The approach is composed of three main steps. First, the old and new code are compared to each other in order to identify if there are requirements-affecting changes. If such changes are detected, then terms describing the change are extracted from the code and are traced to the requirements in order to identify the parts that are likely to be impacted. Finally, the impacted requirements are displayed to the maintainer, who can then update them and save the changes. To evaluate our approach, we developed a set of tools that we used to run the approach on two case studies, namely AquaLush and iTrust. Our approach succeeded to identify between 70% and 100% of the outdated requirements while filtering out more that 80% of the non-impacted requirements in the specification. Automatically identifying the requirements that are likely to be impacted after each source code change is expected to reduce the time and effort needed for updating requirements. Thus, it should also encourage maintainers to regularly update the specification.

For future work, there are two main directions we would like to explore. The first direction aims at improving the tracing approach. We would like to do so by extending the approach so that weights are given to the keywords used for the tracing. The weights will

depend on the source of the keyword, so that keywords extracted from an added method gets a higher weight than those obtained from the call hierarchy. The tracing can also be improved by using more elaborate tracing techniques, which combine information retrieval with machine learning or analyst feedback.

The second direction of the future work is about evaluating the usefulness of the approach for maintainers. For this we plan to conduct a controlled experiment, where we compare the time needed to do the maintenance task, the correctness of the update and the confidence of the maintainer about it when using and when not using our approach. To conduct such an experiment, we need a user-friendly version of the tool that nicely displays the requirements that are likely to be outdated to the maintainer.

Chapter 4

An Automated Hint Generation Approach for Supporting the Evolution of Requirements Specifications

Original publication:
An Automated Hint Generation Approach for Supporting the Evolution of Requirements Specifications
E. Ben Charrada and M. Glinz
Joint ERCIM Workshop on Software Evolution and the International Workshop on Principles of Software Evolution 2010

Abstract

Updating the requirements specification during software evolution is a manual and expensive task. Therefore, software engineers usually

choose to apply modifications directly to the code and leave the requirements unchanged. This leads to the loss of the knowledge contained in the requirements documents and thus limits the evolvability of a software system. In this paper, we propose to employ the co-evolution of the code and its test suite to preserve or restore the alignment between implementation and requirements: when a change has been applied to the code, subsequent changes in the test suite as well as failing tests are analysed and used to automatically generate hints about the affected requirements and how they should be changed. These hints support the engineer in maintaining the requirements specification and thus ease the further evolution of the software system.

4.1 Introduction

Documentation associated with software is considered to be part of the software itself. Among various types of software documents, the requirements specification plays a key role for maintenance and evolution: first, requirements facilitate program comprehension, which is a crucial part of the evolution process. In fact, requirements give a high-level view of the functioning of a system which is easier to understand than code. Requirements also provide the rationale behind an implementation. Understanding how a system works from the structure of its parts only is very complex because the purpose or intent is omitted in the implementation [Lev00]. Second, if the rationale is missing, important design and implementation decisions can be inadvertently undone during maintenance. Third, requirements specifications are accessible to all stakeholders as they do not require a technical background to be understood. Thus they can be used to discuss and negotiate change with stakeholders. To remain useful, the requirements specification has to be maintained when the software system evolves. However, updating the requirements specification is a manual task. In the case of large requirements specifications, it is time-consuming and error-prone. Therefore, maintainers usually choose to apply modifications to source code directly and leave the requirements specification unchanged [You05]. When the software knowledge contained in the requirements specification is lost, it becomes increasingly difficult to apply well-considered changes to the software [BR00].

In this paper we present a technique for automatically generating hints that guide the maintainers to efficiently update the require-

ments specification when the source code is modified. Our idea is based on using modifications in tests to get hints about changes in requirements.

The remainder of the paper is organized as follows. In the next section we motivate our work by presenting the limitations of current requirements update techniques. Then we present our idea of using tests as intermediate between requirements and implementation in Section 4.3. In Section 4.4 we present a classification of software change and discuss the type of hints needed in each case. The approach for hint generation is detailed in Section 4.5. The current state of work is presented in Section 4.6 while related work is discussed in Section 4.7.

4.2 Limitations of Current Requirements Update Techniques

Updating requirements specifications is a manual and expensive task. Ideally, when software engineers receive a change request, they should analyse the impact of the change on requirements, update the specification and then modify the implementation. This task is time-consuming because engineers have to do impact analysis twice: once at the requirements level and then at the source code level. In fact, as source code includes many details which are not present in the specification, analysing the change impact on requirements only is not sufficient to detect all the parts of the code that need to be updated. Therefore, engineers usually choose

to do all the maintenance work (analysis and modification) at the source code level directly and leave requirements specifications unchanged.

Another option to manage requirements evolution is to modify the implementation first, then propagate changes backward to the requirements specification. In this case, it is essential to have dependable traceability links between the requirements specification and source code. This type of traceability links, however, is difficult to define and to maintain. This is mainly due to the difference in structure between requirements and implementation [CHOT99]: requirements usually represent end-user needs, while the implementation reflects many design and implementation details. There is also the problem of scattering (the implementation of a requirement is distributed over many classes) and tangling (one class contributes to the implementation of many requirements). This results in a very large number of links and makes the task of change propagation time-consuming and error-prone.

Another limitation of this approach is that modifications in source code are not always related to changes in requirements (e.g., refactorings). This makes the change propagation task more complex. Consequently, requirements documents are usually not maintained and briefly become obsolete and unreliable.

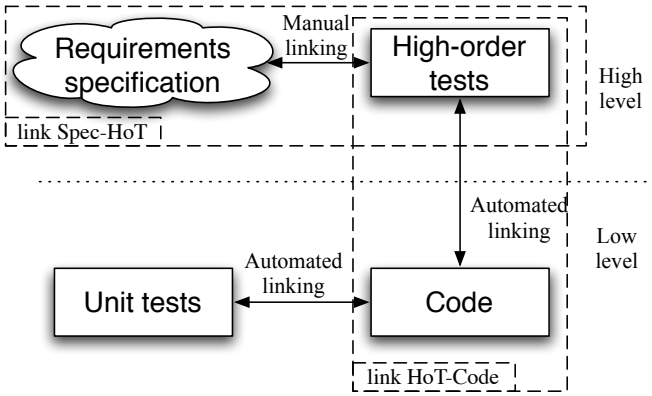

Figure 4.1: Relating requirements and code using high-order tests

4.3 Using Tests to Link Requirements and Implementation

In this paper we present a new approach for supporting the update of the requirements document during software evolution. Our approach builds on two observations:

- Tests are usually maintained with the implementation. For example, Lethbridge et al. [LSF03] found that testing and quality documents are usually updated within a few days after changes are applied to a software system.
- Tests meant to check the external behaviour of the system are derived from requirements thus changes in these tests are usually related to changes in requirements.

Our basic idea is to use the test suite as an intermediate between requirements and implementation during software maintenance and evolution (Figure 4.1). We focus on tests meant to check the external behaviour of the system (e.g., acceptance and system tests). In the rest of the paper we use the term high-order tests, which was introduced by Myers [Mye76] to refer to this type of tests. We analyse modifications applied to the test suite when the source code has been changed and use them to generate hints about changes in the requirements. Guided by these hints, a maintainer can update the requirements specification with little additional effort.

As high-order tests are derived from requirements, defining traceability links between the requirements specification and these tests is straightforward and can be done manually (Figure 4.1: Link spec-HoT). On the other hand, we can obtain the relation between tests and source code automatically when tests are executed against the implementation (Figure 4.1: Link HoT-Code). This makes high-order tests suitable for being an intermediate for propagating changes from implementation to requirements.

In the rest of the paper we assume that the following two conditions are satisfied in the considered software projects: (1) existence of a well-maintained high-order test suite with good requirements coverage and (2) existence of dependable traceability links between requirements and high-order tests.

4.4 Classification of Software Change

In this section we explore different types of software change and discuss the types of hints needed for each of these changes. We base our work on the classification of software maintenance and evolution developed by Chapin et al. [CHK+01]. They distinguish twelve types of software change which are grouped into four clusters: (1) *support interface*, (2) *documentation*, (3) *software properties* and (4) *business rules*. In clusters (1) and (2), we find all maintenance tasks that do not affect source code. Examples of maintenance tasks in these clusters are software evaluation (*support interface*) and documentation update (*documentation*). These tasks affect neither the software implementation nor its tests, thus we do not address them in our work. Table 4.1 contains changes in *software properties* and *business rules*.

Table 4.1: Types of software maintenance

Cluster	Type of maintenance	Do we consider it?
Software properties	Groomative	No
	Preventive	No
	Performance	Yes
	Adaptive	No
Business rules	Reductive	Yes
	Corrective	Yes
	Enhancive	Yes

In the next paragraphs, we go through these seven maintenance types and briefly discuss the type of hint we expect to provide. The approach for hint generation is detailed in Section 4.5.

4.4.1 Reductive Maintenance

A maintenance task is considered as reductive when it removes or limits existing system functionalities. If a functionality is suppressed, tests meant to check the functionality are removed, otherwise they fail. To update the specification, engineers need to know the requirements that have to be removed.

4.4.2 Corrective Maintenance

The goal of corrective maintenance is to fix existing functionalities or make them more precise. Corrective maintenance improves the conformance of the system to the specified requirements; thus it does not imply changes in the requirements document. At the test level, we might add a few tests to check that a fix works correctly. Adding new tests can also be due to the addition of a new functionality (enhancive maintenance). Our hint generation approach needs to differentiate between these two types of maintenance. This point is further discussed in Section 4.5.3.

4.4.3 Enhancive Maintenance

Enhancive maintenance is the most common type of maintenance in the business rules cluster. It includes replacing, adding and extending system functionalities. We treat the cases of addition and replacement separately.

Replacing functionality. When a functionality is modified or replaced, tests related to the functionality have to be modified correspondingly because they would fail otherwise. The hint we expect here is the identification of the requirements to be updated in the specification.

Adding functionality. Addition of functionality is usually followed by the addition of tests covering the new functionality. To update the requirements specification when new functionality is added, the maintainer needs to know what the new requirements are about. It is also interesting to know the relation between the new requirements and old ones because this gives information about the context of the new requirements and helps understanding them. This might also be helpful when establishing the traceability links between the new tests and the requirements specification.

4.4.4 Groomative and Preventive Maintenances

Both groomative and preventive maintenance affect the maintainability of the software. Although maintainability might be required by stakeholders, it is usually not possible to test it. Therefore these types of maintenance have no effect on the test suite and are not considered in our work.

4.4.5 Performance Maintenance

Performance maintenance changes the system performance, which is a non-functional requirement. Similarly to the replacement of functionalities (enhancive maintenance), the hint we expect here is the identification of the requirements affected by a change.

4.4.6 Adaptive Maintenance

A maintenance task is adaptive when it affects the technologies or resources used by the system. Adaptive maintenance might not require modifications in tests. For example, it is usually possible to check that a system works correctly on two different platforms, by running the same tests on each of these platforms. Therefore, generating automated hints about adaptive maintenance from tests might not be feasible. In our current research we do not consider adaptive maintenance, but we will address it in future work.

4.5 Approach for Hint Generation

In this section we present our approach for hint generation. We detail the approach in the case of enhancive maintenance, which is the most common type of software change among the ones we are considering [CHK+01]. Generating hints for other types of maintenance is discussed briefly in Section 4.5.3 and will be elaborated in future work.

4.5.1 Automatically Identifying Requirements Affected by Change

In this section we define rules for generating the set of requirements affected by a change. The rules are based on the modifications applied to tests and on the traceability links between tests and requirements. We formulate the addressed problem as follows:

R is a set of requirements, T is a set of high-order tests and L a set of links relating elements in R to elements in T. After applying change to the software, T_c and R_c are subsets of T and R containing elements affected by the change. Our goal is to derive R_c based on T_c and L.

The main challenge faced here is that we usually do not have a one-to-one mapping between requirements and tests. In fact, one requirement might be checked by many tests and one test might check many requirements. An intuitive way to get the set of requirements that need to be updated is to look for all requirements related to tests that have been changed and include them in R_c. The algorithm is the following:

for $t \in T_c$
 for $r \in R$
  if r is related to t then $R_c \leftarrow R_c \cup r$

This algorithm is very simple, but yields results having low precision: it generates many false positives resulting in much unnecessary work for the maintainer. A more sophisticated way to address

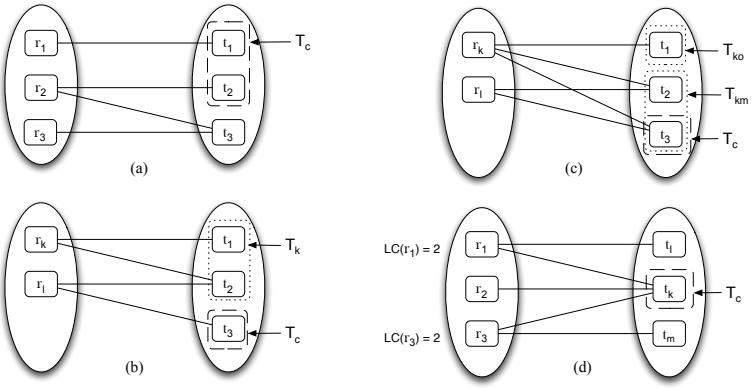

Figure 4.2: Identifying affected requirements

the problem is to evaluate, after each change, the likelihood of a requirement to be affected. We have developed four rules for estimating this likelihood. The likelihood for a requirement to change (LC(r)) is represented using a number from 1 to 5, where each of these numbers represent one of the following categories:

5- The requirement is affected by the change
4- The requirement is likely to be affected by the change
3- We cannot decide about the requirement
2- The requirement is likely not to be affected by the change
1- The requirement is not affected by the change

Rule A

If a test $t \in T_c$ is related to only one requirement r, then r is affected by change: LC(r)=5.

This case is illustrated in Figure 4.2 (a), where LC(r_1)=LC(r_2)=5.

Rule B

If requirement r_k is related to a set of tests T_k, and $T_k \cap T_c = \emptyset$ then LC(r_k)=1

Figure 4.2 (b) is an illustration of the case.

Rule C

Consider a requirement r_k related to a set of tests $T_k = T_{km} \cup T_{ko}$ where T_{ko} are tests related to r_k only and T_{km} are tests related to r_k and other requirements (Figure 4.2 (c)). *If $T_{ko} \cap T_c = \emptyset$ and $T_{km} \cap T_c \neq \emptyset$ then r_k is not likely to change: LC(r_k)=2*

The rationale behind this rule is that if r_k changes, both T_{km} and T_{ko} should be affected by this change. If only T_{km} changes, then the change in tests is probably related to changes in other requirements (r_l in the case of Figure 4.2 (c)).

Rule D

Consider a set of requirements R_k = (r_i,..,r_m,..,r_p) related to a test $t_k \in T_c$. If all requirements other than r_m are not affected by change (LC(r)¡3 for all r such as $r \in R_k$ and $r \neq r_m$) then r_m is likely to change: LC(r_m)=4.

The rule is illustrated in Figure 4.2 (d). Modifications in test t_k are due to changes in r_1, r_2 and/or r_3. As r_1 and r_3 are not likely to change (we calculate their likelihood to change by applying the previous rule, but the values might also be set manually by the maintainer) then the modification is probably due to change in r_2: LC(r_2)=4.

Applying the Rules
As the rules are not mutually exclusive, the order of their application matters. We apply the rules in the following order: A, B, C, D. If none of the rules apply for a requirement r, then we cannot decide whether r needs to be modified or not: LC(r)=3.

4.5.2 Automatically Generating Hints About New Requirements

When a new functionality is added to the system, we need to know what the functionality is about so that we can update the specification. We intend to get this information from the source code. In most cases, source code contains information about system functionalities and behaviour. For example, method names usually reflect the purpose of the method, and its documentation contains extended details about the role of the method. However, as source code is usually huge, relevant information about system functionalities may be buried in a large number of design and implementation details. The challenge here is to extract the right information from the source code. We will exploit the information in the source code by analysing test execution traces. By test execution traces, we mean the methods that are called when the test is running. Tracing all methods does not work because it would yield a very large number of irrelevant traces. Therefore we do a selective tracing: we only trace methods having names that are similar to words present in the newly added tests.

We illustrate our idea using an example of a simple library management system that manages borrowing and returning books.

Suppose that maintainers add a new functionality for sending reminders to borrowers for returning books. Then they update the test suite by adding a test to check the functionality. In the test we will probably find the words "send" and "reminder". When analysing the test execution traces, we might find a method called "sendReminder". The code and documentation associated to this method probably contain information about when and how this method is used.

Another relevant hint we can extract from execution traces is the relation between the new requirement and old ones. We propose to do this like in [Egy03]. We consider overlaps between execution traces as indicators about relations between requirements: if tests t_1 and t_2 are related to requirements r_1 and r_2, respectively, and if execution traces of t_1 and t_2 overlap, then we deduce that r_1 and r_2 are related to each other.

4.5.3 Generating Hints for Other Maintenance Types

In the case of reductive and performance maintenance, identifying the requirements that need to be removed or modified can be done by defining identification rules in a way which is similar to what we did for functionality replacing (Section 4.5.1). We might also use execution trace analysis and look for key methods that have disappeared from the traces to identify functionalities removed during reductive maintenance.

When new tests are added, we need to differentiate between corrective and enhancive maintenance. We to do this by analysing the test coverage: in the case of corrective maintenance, the coverage of the added tests is very similar to existing tests that are meant to test the same functionality. On the other hand, when new functionality is added, new tests cover newly added code.

4.6 State of Work

We have applied our approach to a simple library management system that we have developed as a testbed. First results show that the approach works, but the yield is still too low: we have many cases where the likelihood of change evaluates to three, i.e. it is not possible to give a hint. We are currently exploring the effect of considering the type of change applied to the test suite and the test coverage in improving the decisiveness of our rules. Our current work focuses mainly on enhancive maintenance. Other types of maintenance will be addressed by elaborating the ideas presented in Section 4.5.3: new rules have to be defined to generate relevant hints for each type. Concerning the type of requirements we are considering, we started by analysing changes in textual requirements specifications. We intend to cover also requirements expressed in modelling languages such as ADORA [GBJ02] or UML in the future.

The validation of our approach will be based on a case study. We will measure the precision and recall of the generated hints as well as the usefulness of these hints for updating requirements.

4.7 Related Work

Our work relies on traceability links between requirements and tests. Much research has been done in the field of defining and updating traceability links. Antoniol et al. [ACC+02] and Hayes et al. [HDS06] developed methods based on information retrieval models to generate traceability links between source code and documentation automatically. Egyed [Egy03] uses trace analysis to semi-automatically find dependencies between requirements and generate links. Mäder and Gotel [MGP08] developed an approach to update traceability links for UML models during software maintenance.

Our research subject is related to the problem of co-evolution of artifacts, which is also addressed in the work of Mens et al. [MKPW06] and Reiss [Rei02]. However, these works focus on the co-evolution of design and implementation, while our focus is on requirements and implementation.

In order to solve the problem of obsolete or non-existent requirements documents, Yu et al. [YWM+05] propose to reverse engineer requirements goal models from code. The main limitation of reverse engineering methods is that the generated artifacts are either imprecise or incomplete [CDP07]. Thus we expect our approach to provide more dependable requirements.

4.8 Conclusion

We presented an automated approach to support requirements specification maintenance during software evolution. Modifications applied to tests are analysed and used to generate hints about changes in requirements. These hints are expected to decrease the costs needed for updating requirements and thus help preserving the valuable knowledge contained in them. We focused in this paper on identifying requirements affected by change and getting hints about newly added requirements in the case of enhancive maintenance. Other types of changes, as well as an evaluation of the efficiency and effectiveness of our approach, will be addressed in future work.

Chapter 5

Towards a Benchmark for Traceability

Original publication:
Towards a Benchmark for Traceability
E. Ben Charrada, D. Caspar, C. Jeanneret and M. Glinz
Joint ERCIM Workshop on Software Evolution and the International Workshop on Principles of Software Evolution 2011

Abstract

Rigorously evaluating and comparing traceability link generation techniques is a challenging task. In fact, traceability is still expensive to implement and it is therefore difficult to find a complete case study that includes both a rich set of artifacts and traceability links among them. Consequently, researchers usually have to create their own case studies by taking a number of existing artifacts and

creating traceability links for them. There are two major issues related to the creation of ones' own example. First, creating a meaningful case study is time consuming. Second the created case usually covers a limited set of artifacts and has a limited applicability (e.g., a case with traces from high-level requirements to low-level requirements cannot be used to evaluate traceability techniques that are meant to generate links from documentation to source code). We propose a benchmark for traceability that includes all artifacts that are typically produced during the development of a software system and with end-to-end traceability linking. The benchmark is based on an irrigation system that was elaborated in a book about software design. The main task considered by the benchmark is the generation of traceability links among different types of software artifacts. Such a traceability benchmark will help advance research in this field because it facilitates the evaluation and comparison of traceability techniques and makes the replication of experiments an easy task. As a proof of concept we used the benchmark to evaluate the precision and recall of a link generation technique based on the vector space model. Our results are comparable to those obtained by other researchers using the same technique.

5.1 Introduction

Traceability supports to a great extent the tasks of maintaining and evolving software systems. In fact, it allows tracing the implementation back to the requirements and design documents and thus facilitates the comprehension of the code. Traceability links are also useful for analysing the impact of change and estimating the effort needed for implementing it. Furthermore, traceability links can be used to ensure that all new requirements are implemented and all impacted artifacts are updated.

Although very beneficial, traceability is rarely used because it is expensive to implement and maintain [WN94]. Therefore, in the last years, several researchers have developed techniques and tools for generating traceability links automatically [ACC+02] [HDS06] [HDS+07] [MM03] [CHCGE10] or semi-automatically [Egy03]. The rapid progress in this field of research has increased the need for performing easy and rigorous validations and comparisons of traceability techniques and tools.

To satisfy this need, the traceability community is currently working on the definition of benchmarks for traceability [CHCD+11]. A benchmark is *"a test or set of tests used to compare the performance of alternative tools or techniques"* [SEH03]. It is difficult to acquire meaningful and non-trivial data sets that can be used to create traceability benchmarks [CHCD+11] because there are almost no publicly available projects that include traceability links.

In this paper, we present a candidate benchmark for traceability that includes a rich data set with end-to-end traceability links. We developed the benchmark based on the software for an irrigation system that is published in a book about software design [Fox06]. The benchmark includes all the typical artifacts that are produced during the development of a software system. The main feature of the proposed benchmark is that it provides end-to-end traceability links. As a proof of concept, we used our benchmark to evaluate the results obtained by the Retro traceability link generation [HDS+07], and compared these results to those obtained by other researchers using the same tool or the same technique. The results we obtained were comparable to those obtained by other researchers.

Our paper is structured as follows. In Section 5.2 we give a brief introduction to traceability and discuss the main challenges related to the evaluation of new traceability techniques. Section 5.3 is about the benchmark development: first, we present the rationale for a traceability benchmark, then we explain our benchmark and discuss the properties that such a benchmark should have. We present a proof of concept in Section 5.4, where we explain how we used our benchmark to evaluate the results obtained from a traceability link generation tool. In Section 5.5 we discuss the threats to validity and limitations of our benchmark. The next steps of our work are presented in Section 5.6. Finally we discuss the related work in Section 5.7.

5.2 Background and Motivation

5.2.1 Traceability

Software traceability is defined as *"the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artefacts, and the rationale that explains the form of the artefacts"* [SZ05].

Traceability is used to support different maintenance activities [SZ05]. It is useful for change impact analysis as it allows identifying the parts affected by a change and thus estimating the effort needed for applying the change. It also supports software verification and validation as it allows checking that all requirements have been implemented in the system and that the system satisfies its specification. Program comprehension can be much easier when traceability links are available because developers can easily trace code elements back to the original requirements, which give the rationale behind the implementation. They can also trace code elements back to the design and architecture documents to get a more abstract view of the system.

Much research has been conducted to support the generation, the maintenance and the use of traceability links. Among these three subjects, the generation of links is currently the most active one. The reason is that defining traceability links is still the

most important and the most challenging task. Additionally, maintaining and using traceability links only make sense if the traceability links are defined. Therefore, we focus our work on the generation of links.

Various approaches have been developed to generate traceability links among different types of software artifacts automatically. Most of these approaches are based on Information Retrieval (IR) models such as the probabilistic IR model or the vector space model. The probabilistic IR model computes the probability that two documents are related and uses the calculated probability to rank generated links [ACC+02]. In the Vector Space IR model, the similarity between two documents is calculated as the cosine of the angle between two vectors D_1 and D_2, where the elements of D_1 are the weights of the vocabulary terms in the first document and D_2 the weights of the terms in the second document [HDS06].

Generated links are usually evaluated in terms of precision (*"the number of correct retrieved links (C) divided by C plus the number of retrieved false positives"* [HDS05]) and recall (*"The number of correct retrieved links (C) divided by C plus the number of correct missed links"* [HDS05]).

In order to improve the quality of the generated links, many researchers combined IR techniques with other methods such as analyst feedback [HDS06], machine learning [CHCGE10] or execution tracing [EAAG08]. These methods improved the quality of generated links. However, the precision of generated links is still low when the recall is high.

Figure 5.1: Examples of AquaLush artifacts

5.2.2 Evaluating traceability link generation techniques: The Challenge

As more and more effort is spent to enhance and improve the traceability link generation techniques, the need for a rigorous evaluation of these techniques is increasing. However, it is difficult to find a project that includes a rich set of artifacts and the traceability links among them and which can be used to evaluate link generation techniques. Indeed, as traceability is expensive to implement, almost all publicly available projects have either no or only partial traceability. Projects that do have traceability links are in most cases confidential and cannot be published. Hence, researchers usually develop their own examples to evaluate their approaches. There are two main problems related to developing one's own example. First, the development of a meaningful and large enough example takes time. This distracts researchers from their original goal, namely elaborating effective traceability techniques. Second, the cases are constructed specifically for one particular method or technique; therefore they are not necessarily usable by other researchers in the field. For example, if a researcher develops an example that includes the traceability links between requirements and design documents, the example can only be used for generating links between these two types of artifacts. Whoever would like to evaluate a technique for generating links between design documents and code will have to develop a new example.

Comparing traceability techniques is also an important challenge. The quality of generated links depends considerably on the used

example. In [OGPDL10], the authors obtained 90% recall for a precision of 17% when using the vector space model on the EasyClinic project, while they got 47% recall for the same precision value when applying the same technique on the eTour project. Therefore, the effectiveness of link generation techniques can only be compared when they are applied on the same case.

Consequently, there is an urgent need for developing a case that is both complete and publicly available to the community. The case should include all artifacts that are produced during the development of a software system, as well as the traceability links among these artifacts. Such a case can then serve as a benchmark for traceability. This can be done by defining the tasks that should be performed on the case and the measures that are used to evaluate the effectiveness of the method used to perform the tasks. Constructing a benchmark for traceability will not only solve the problems mentioned above, it will also support advancing the research in the field of traceability because it facilitates the replication of experiments and the comparison of results to each other [SEH03] [Tic98].

In the next section, we present the basic components of a traceability benchmark that includes many artifacts produced during the development of a software system and provides end-to-end traceability links among these artifacts.

5.3 The Benchmark

5.3.1 Is it The Right Time for it?

Sim et al. [SEH03] mention two conditions that need to be satisfied before making attempts for constructing a benchmark for traceability. First, the field of research needs to be mature enough so that the benchmark does not hold back the progress in the community. Second, there must be willingness for collaboration within the community because this facilitates the acceptance of the benchmark and its use.

Attempts to compare traceability approaches indicate that the field is mature enough for the development of a benchmark. In the last years, there have been several studies comparing automated link generation approaches based on different information retrieval techniques [HDS06] [ACC+02] [MM03] [OGPDL10]. Recently, researchers began to use data and cases developed by other laboratories to evaluate their traceability techniques (e.g., in [MM03], the authors evaluated their approach using the data that were developed by other researchers in [ACCDL00]). This facilitates to a great extent the comparison of these techniques.

Many events for traceability illustrate the willingness for collaboration within the community (e.g., the workshop on Traceability in Emerging Forms of Software Engineering and the International symposium on Grand Challenges in Traceability).

An exploration of the status of research in the field of traceability shows that the discipline is beyond the cited preconditions. In fact, the community is already aware that a benchmark for traceability is needed. Indeed, this need has been explicitly mentioned and discussed by a number of researchers in the field [CHDH+06] [DHA07]. The traceability community is also engaged in establishing a research infrastructure that facilitates the running of traceability experiments [CHCD+11]. This infrastructure is expected to include a number of traceability benchmarks.

5.3.2 Creating the Benchmark

AquaLush

We constructed the data of our benchmark based on an existing case study: AquaLush [Fox06]. AquaLush is an irrigation system that uses soil moisture sensors to control the irrigation of the soil. The AquaLush case study is an illustrative example from a book about software design. We chose AquaLush because it includes a rich set of documents that covers several artifacts produced during different development stages. The AquaLush artifacts that we included in our benchmark are the user-level requirements, the use case model, the software requirements specification, the software architecture document, the detailed design document and the source code. The documents are written in natural language and the source code is written in Java. The documents contain a number of diagrams (e.g. class diagrams or statecharts), tables

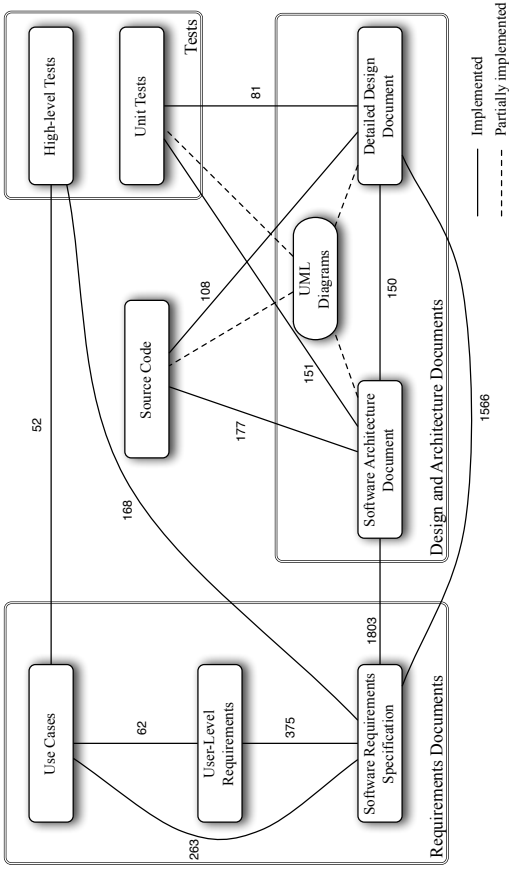

Figure 5.2: The data set and answer set of the benchmark

and GUI screenshots. Figure 5.1 presents some examples of the AquaLush artifacts.

The AquaLush project was developed for pedagogical purposes, therefore it includes documents that reflect the practices lectured in software engineering courses. We found a few inconsistencies among some of the artifacts (e.g. methods mentioned in the design document but not implemented in the code) and a number of bugs in the implementation, but these are typical problems that are likely to be found in any software project. We did not fix these problems, as we did not want to alter any of the project data.

Benchmark Components

The design of our benchmark was inspired by the work of Dekhtyar et al. [DHA07] who decompose a benchmark for traceability into five components: data set, tasks, answer set, measures and software/data format. We detail each of these components in the following paragraphs.

Data Set Our goal was to develop a complete data set that covers the main artifacts produced during the development of a software system. We developed the data set based on the AquaLush case study. We took the existing AquaLush documents, which already cover artifacts from the requirements, design and implementation stages and added some tests in order to cover the testing stage.

We developed 2 types of tests: unit tests and high-level tests. Both kinds of tests are automated and implemented using JUnit[1]. We used the following testing techniques to develop the unit tests [Cas11]: equivalence partitioning, boundary-value analysis and branch coverage. We also considered the Liskov Substitution Principle [LW93] which states that if a type B is a subtype of another type A in an object-oriented program, then it should be possible to replace objects of type A with objects of type B each time an object of type A is used without having to change the rest of the program. The unit tests cover all classes except those in the user interface layer and in the start-up layer. The high-level tests have been developed according to the use cases available in the requirements documents and they cover both the basic and alternative flows of the use cases. Table 5.1 present an estimated size of the documents in the data set.

Table 5.1: Size estimation of the benchmark data set

Document	Size	
User-level requirements	49 statements	599 words
Use Cases (one use case includes several extensions)	8 use cases	2075 words
Software requirements specification	372 statements	7370 words
Software architecture document	109 statements	5497 words
Detailed design document	64 statements	3803 words
Diagrams	23 diagrams	
Source code	75 classes	11 KLOC
Tests	93 classes	15 KLOC

[1] http://www.junit.org/

To manage the artifacts, we used two tools: DOORS[2] and Rhapsody[3]. Textual documents, tables and screenshots were entered in DOORS. Each message is entered as one element in DOORS. If a title (or a subtitle) has only one message below it then the title and the message are merged in one element. In the opposite case, the title is considered as one element. The UML diagrams were extracted from the architecture and design documents and were entered in Rhapsody.

Tasks There are various traceability related tasks that can be performed using the AquaLush data set, including the use of traceability links among the artifacts for identifying the impact of a change or for propagating changes among the artifacts. In this work, we focus on defining traceability links.

We intend to evaluate and compare the effectiveness of methods and tools in generating complete and correct traceability links among different types of artifacts. We are especially interested in the generation of end-to-end vertical traceability linking (1) the requirements to the design and architecture documents and(2) the architecture documents to the source code and tests (Figure 5.2). We also consider the generation of links between high-level tests and requirements because these high-level tests are meant to check that the requirements are satisfied and are therefore related to them. The resulting links allow tracing any artifact to any other either directly or by going through an intermediate artifact. For example,

[2]http://www.telelogic.com/products/doors/
[3]http://www-01.ibm.com/software/awdtools/rhapsody/

it is possible to trace elements from the requirements specification to elements in the code by first finding elements in the software architecture document related to the requirements and the finding elements related to these architectural elements. The proposed tasks do not depend on any specific traceability tool or technique and they can be achieved manually or automatically.

Answer Set The answer set (or ground truth) is the set of correct traceability links that relate the AquaLush artifacts to each other. We defined these links manually among the main AquaLush artifacts as presented in Figure 5.2. The numbers in Figure 5.2 are the count of traceability links that we have defined. We used DOORS to define links among all the textual artifacts, tables and pictures. For UML diagrams, we used Rhapsody, which allows us to link internal elements of the diagrams (e.g. link a class or a method). We also used DOORS and Rhapsody to define links pointing to the source code. Links to the source code point either to a classes or to a package.

We defined our links according to the following rule: an element B is related to another element A if B is derived from A or if B gives additional and useful information about A.

There were a few elements (methods, constructors and classes) that have been mentioned in the architecture and design documents but which were not implemented in the code. As we did not want to modify any of the existing AquaLush artifacts, we did not link these elements to the code.

Measures To evaluate the effectiveness and efficiency of a traceability techniques, we use three measures: precision, recall and time. Precision and recall (see Section 5.2.1) are frequently used to evaluate link generation technique. The time measure is meant to quantify the time needed by a given technique for generating the traceability links.

Software/Data Format The format of the benchmark data should be easy to use and independent of any specific tool. As DOORS and Rhapsody documents do not satisfy this property, we exported all textual data into HTML format and all UML diagrams into XMI format. There are two advantages for using HTML. First, it allows browsing the documents easily and navigating among related artifacts using simple clicks. Second, HTML documents are well structured and any specific information can easily be extracted from them. For example, it is possible to extract the traceability links and create a traceability matrix out of them using a small piece of code. We used XMI for UML diagrams because it is a standard format for exchanging UML diagrams. UML diagrams are available as images too. We also provide the Doors and Rhapsody documents for those who would prefer to use these tools.

5.3.3 Desiderata for a Benchmark

To be successful, a benchmark should satisfy some properties. Sim et al. [SEH03] identified seven desiderata for a benchmark in

the field of software engineering: accessibility, affordability, clarity, relevance, solvability, portability and scalability. Dekhtyar et al. [DHA07] identified five additional requirements for a traceability benchmark: support for traceability in multiple software engineering fields, independence of methodology, ground truth, accuracy testing and scalability testing. In the remainder of this section, we discuss to which extent our benchmark meets these characteristics.

Accessibility The benchmark data need to be easy to obtain and easy to use. To satisfy this property, we published all the data of the benchmark at:
http://www.ifi.uzh.ch/rerg/research/aqualush/.
The data includes all the AquaLush artifacts mentioned in the previous section and the traceability links relating the artifacts to each other. Anyone can get the data, use it and eventually extend it to meet other requirements.

Relevance The benchmark tasks should be representative of the typical traceability operations that are performed in real life. This condition is satisfied because the AquaLush artifacts are representative of the main artifacts produced during the development of a software system. The task of generating traceability links is also a typical operation that has been addressed by many researchers in the traceability field.

Clarity AquaLush is an illustrative example from a book about software design. Therefore, it is easy to understand and is self-contained. The benchmark task (generate traceability links among various artifacts) is a classical and clear task that has already been performed by many researchers. The benchmark measures (precision, recall and time) are also simple and classical measures that have been used by researchers in the field.

Affordability The benchmark must not be difficult or expensive to run, because otherwise researchers will not use it. The AquaLush project is not large and the link generation task is clear, therefore the benchmark is easy to use. Before using the benchmark, people may need to modify the format of the artifacts and the traceability links in such a way that they can be used by the tools or approaches they are using. This task can easily be automated in most of the cases.

Solvability It should be possible to produce a good solution for the benchmark task. Our traceability benchmark is solvable and the solution, which is the set of traceability links among the different artifacts, is provided within the benchmark.

Portability The benchmark should be portable to different tools and techniques. Both the AquaLush artifacts and the traceability links among the artifacts do not depend on any specific tool or technique. The data is available in a standard format (HTML and XMI). Therfore, it can be used to evaluate any tracing tool or technique.

Scalability Scalability might be the major limitation of this benchmark. While the AquaLush project includes many types of artifacts, it is not a large project and thus it is not representative of large systems. To improve the scalability of the benchmark, AquaLush may be extended with new features. This extension is left for future work.

Support of multiple SE fields The benchmark should be rich enough to support various tasks related to tracing. Currently, we only consider the generation of links among various types of artifacts. However, as our benchmark includes a complete data set with end-to-end traceability, it can be used for evaluating tasks from other software engineering processes such as verification and validation or maintenance and evolution. For example, analysing the impact of changing one requirement on the rest of the artifacts is a task in software maintenance that can be evaluated with our benchmark.

Independence of methodology The benchmark should not depend on any specific tracing tool or technique. This requirement is satisfied by our benchmark. In fact, all tracing methods, whether manual, semi-automated or automated, can be used to solve the task of defining traceability links among the AquaLush artifacts.

Ground truth The true answer for each of the benchmark tasks should be provided. In our case, the true answer is the set of

traceability links that relate the AquaLush artifacts to each other. We have defined these traceability links and we provide them with the benchmark.

Accuracy testing The benchmark should allow evaluating the accuracy of tracing techniques. In our benchmark, we assess the accuracy through the precision and recall measures.

5.4 Proof of Concept

5.4.1 Experiment

As a first application of our benchmark, we used an information retrieval tool to generate traceability links among some artifacts in the benchmark and we compared the generated links with the ground truth (that is, the traceability links we defined manually). We then compared the results we obtained in term of precision and recall to the results published by other researchers who used the same traceability technique on other case studies. We considered two tasks: generating links from the user-level requirements (ULR) to the software requirements specification (SRS) and generating links from the software architecture document (SArch) to the code.

The goal of this experiment is to answer the following questions:

Q1: Are the results (precision/recall) obtained when generating traceability links for the AquaLush project similar to those obtained by other researchers using the same techniques on other cases? What conclusion can we derive from these results concerning the relevance of our benchmark?

Q2: Do we get similar results when we use the same traceability tool to extract links (1) among two documents written in natural language and (2) among a document written in natural language and source code?

To run our experiment we used a research tool for generating traceability links: Retro (REquirements TRacing On target) [HDS+07]. Retro provides a number of IR methods that can be used for recovering links. It also allows filtering candidate links by specifying a threshold value for the traces that should be considered: if the threshold is 0.15, then only the links having relevance greater than 0.15 are kept. Analysts can enter feedback Retro to improve the quality of generated links, but in our experiment we did not use the feedback feature.

Retro takes two lists of textual files as input: the high-level documents and the low-level documents. Therefore, we had to split the documents (ULR, SRS and SArch) into small sub-documents, where each sub-document contains one element that should be traced. For the source code, we considered each class as a single document. As Retro only considers textual documents, we removed all pictures and diagrams from the document. We also extracted the content of each tables into text files.

We generated links using the default tracing method in Retro, which is the vector space retrieval with tf-idf (term frequency - inverse document frequency) term weighting. Then we filter links with different threshold values and see the effect of the filtering on precision and recall. The resulting links are automatically compared with the ground truth of our benchmark.

When evaluating the links generated among the architecture document and the code, we were obliged to alter our manual traceability links (i.e., the answer set). The reason is that a number of manual links (32 links in this case) were pointing to a whole package and not single classes (e.g. general statement about GUI were linked to the UI package). However, as Retro does not consider the hierarchical structure of documents, it only generated links pointing to classes. To make the comparison of links possible, we split the links pointing to a package into several links pointing to each of the classes within the package as illustrated in Figure 5.3. We call the experiment we run with these links *AquaLush (+)*. We also run a second experiment (*AquaLush (-)*) where we neglected all the links pointing to packages.

Managing the titles and subtitles was also a challenge. Titles contain relevant keywords related to the statements under them. But existing traceability links do not take documents' structure and titles into consideration. Therefore, in this experiment, we deleted all elements that contain a title only.

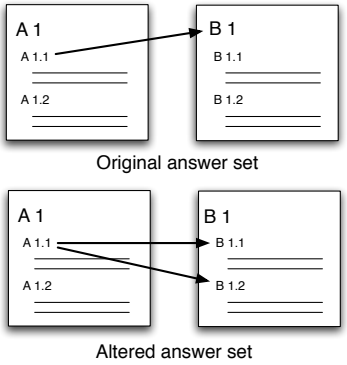

Figure 5.3: Splitting traceability links through the hierarchy

5.4.2 Results

In this section, we compare the precision and recall we obtained with AquaLush to those published in [OGPDL10] and in [SHD05]. We also compared the results obtained in the two link generation tasks.

In [SHD05], Sundaram et al. used Retro to generate traceability links between high and low-level requirements for two data sets: MODIS and CM-1. MODIS contains 19 high-level requirements and 49 low-level requirements. There are 41 links between high-level and low-level requirements. CM-1 has 220 high-level requirements, 235 low-level requirements and there are 361 links among these requirements.

We took the results they obtained using tf-idf and no analyst feedback and compared them to the precision and recall we obtained when generating links from the user-level requirements to

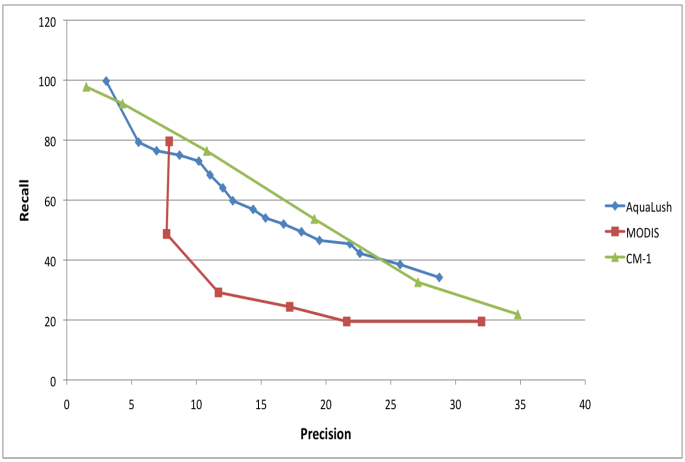

Figure 5.4: Precision and recall for traceability links generated from "user-level requirements" to "software requirements specification"

the software requirements specification of AquaLush. As we used the same IR method, the same tool and similar types of artifacts as in [SHD05], we expected to obtain results that are comparable to each other. The results are reported in Figure 5.4. The precision and recall obtained for CM-1 and for AquaLush are very similar.

Comparing the results obtained from generating links between source code and architecture documents with external results was more challenging. In fact we could not find results for these types of documents, but we found results for using IR methods to get links between code and other types of artifacts.

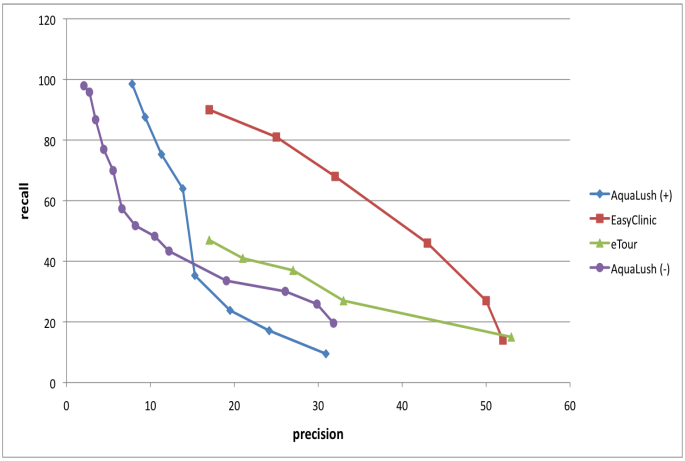

Figure 5.5: Precision and recall for traceability links generated from "architecture document" to "source code"

In [OGPDL10], Oliveto et al. compared different IR techniques for generating links among use cases and source code for two data sets: EasyClinic and eTour. EasyClinic has 30 use cases, 47 classes and 93 correct links while eTour has 58 use cases, 116 classes and 336 correct links. Among the various results presented in the paper, we considered those obtained with the Vector Space Model. These results are reported in Figure 5.5.

The recall obtained with both AquaLush experiments is lower than the one obtained with EasyClinic and eTour for precision values that are over 17% (no recall values were reported in [OGPDL10] for higher precision). The difference is however not huge for the case of AquaLush (-) and eTour.

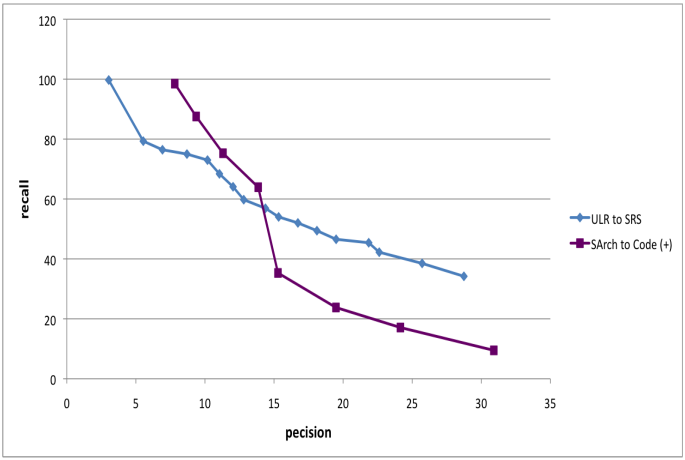

Figure 5.6: Precision and recall for traceability links generated from "user-level requirements" to "software requirements specification" and from "architecture document" to "source code"

In Figure 5.6, we report the results we obtained for both link generation tasks. Globally, the results we obtained when generating links from ULR (user-level requirements) to the SRS (software requirements specification) are comparable to those obtained when generating links between SArch (software architecture document) and code.

Answering Q1 The results we obtained when generating traceability links among ULR and SRS are very similar to those obtained by Sundaram et al. [SHD05] when using the same tool on similar types of artifacts from the project CM-1. This similarity is a positive indicator about the fitness of our benchmark for evaluating

traceability generation techniques, because the developers of the Retro tool could have obtained similar results if they used our benchmark for validation.

The precision we obtained for generating links among SArch and code are not as good as those obtained by Oliveto et al. in [OGPDL10]. There are various explanations for the lower precision we obtained in this experiment. First, the use of different cases with different types of documents (use cases in [OGPDL10] vs. architecture document in our experiment) is likely to give different results. Second, we did not use the same tool (no tool was mentioned in [OGPDL10]), so the technique they used for generating links may be different from ours. In fact they do not mention which term weighting was used therefore it is possible that it is not the one we used (tf-idf). The text normalization may also be different, as code and textual documents are normalized in different ways. Finally, our results are probably affected by the modifications made to the manual links (see Section 5.4.1) in order to make the comparison possible.

Answering Q2 Retro performs similarly when generating traces among two documents written in natural language and when generating traces among a document in natural language and source code. While we can make no reliable conclusions about the performance of Retro based on a single experiment, the experiment suggests that Retro could achieve comparable results when used for tracing documents written in natural language and when used for source code.

5.4.3 Lessons Learned

In this section, we present a number of problems that we faced during the development of our benchmark and how we mitigated them.

Rules for defining links When defining the traceability links, deciding whether to consider two elements were related or not was sometimes difficult. In the literature, we could not find guidelines about how to define traceability links within a project. Therefore, we defined our own guidelines: we considered two elements A and B as related only if B is derived from A or if A gives additional and useful information about B. There may be other ways to define traceability links among artifacts depending on the purpose behind their implementation. We encourage traceability researchers to explore what kinds of links are most useful and how these links should be defined.

Support for hierarchical documents Current techniques for generating traceability links do not support the hierarchical structure of documents. For example, they do not allow linking one element to a whole section or subsection in a document. They also do not take titles and subtitles into consideration. In the answer set of the benchmark, there are links pointing from elements in the architecture documents to packages in the code. It was not possible to directly compare these links to those generated by Retro because the links generated by Retro only point to classes.

To overcome this problem, we split the links pointing to a certain package into several links pointing to each of the classes within the package (see Section 5.4.1). Supporting the hierarchical structure of documents is an interesting direction for future research.

5.5 Threats to Validity

In this section, we discuss the threats to the validity of our benchmark.

Construct Validity To evaluate the effectiveness of traceability approaches, we use two classical measures that have been extensively used to evaluate IR based approaches: precision and recall. However, these measures may not cover all the strengths and weaknesses of a tracing method or tool. Therefore, it would be interesting to consider additional measures that assess other dimensions of the traceability tool or method, such as the usefulness of the generated links for program comprehension.

Internal validity The AquaLush artifacts used for creating the benchmark were developed by a third party who was not aiming at using them for traceability purposes. This reduces the risk of having artifacts that are tailored for facilitating the generation of traceability links among them or artifacts that are adapted to some special traceability tool or technique.

External validity AquaLush is a relatively small project which has high-quality artifacts. It is therefore not representative of real large software projects which may have incomplete and low-quality artifacts.

The evaluation of the effectiveness of a tool or technique depends on the example used for the evaluation. In most cases, we get different results when we apply the same tool or technique on different case studies. Therefore, a single case study is not enough to draw generalizable conclusions. A good benchmark should include documents that are representative of different types of projects. Thus, it is important to expand our benchmark in the future by adding other types of projects.

Other limitations A major limitation of our benchmark is that the traceability links we defined (i.e., the ground truth) have not been validated. We intend to contact the original developer of AquaLush to evaluate the relevance of the links.

5.6 Next Steps

We are currently working on finalizing the answer set by completing the traceability links among the artifacts as presented in Figure 5.2. For future work, we intend (1) to use the benchmark to compare different traceability links and tools and (2) to extend the benchmark to cover other traceability related tasks.

Using the benchmark We will use our benchmark to compare the effectiveness of different traceability link generation techniques and tools. The goal of the experiment will be to find which tool or technique is most efficient for each type of document. Each technique/tool will be used to generate links among the different AquaLush documents and the results will be compared to each other.

Extending the Benchmark We intend to support the following three tasks in the future: Analysing the impact of change, tracing bug reports and updating traceability links. The goal of the impact analysis task is to identify all the artifacts affected by a given change. To cover this task, we will define a number of changes (like bug fixes, changes in the external behaviour of the system or changes in the design) and identify all the artifacts affected by the change. We will also define measures that estimate the time needed for performing the analysis and the correctness of the obtained results.

The bug-tracing task is about generating traceability links among bug reports and source code. For this task, we will extend AquaLush with some bug reports and traceability links from the bug reports to the source code.

Updating traceability links is a challenging task that can be evaluated using our benchmark. We will create a second release of the AquaLush artifacts and define traceability links among them. The task will then be to identify all links affected by the changes in the artefacts and update these links.

5.7 Related Work

In this section, we present some case studies used to evaluate traceability generation techniques. To the best of our knowledge, none of these cases cover all the artifacts that are covered by our benchmark and provide end-to-end traceability among the artifacts.

Hayes at al. [HDS06] use data sets obtained from two NASA projects: CM-1 and MODIS. These data sets only cover high-level and low-level requirements. In [OGPDL10], Oliveto et al. used data sets from the Etour and the EasyClinic projects. Etour only includes use cases and code classes. EasyClinic includes uses cases, a textual representation of interaction diagrams, source code and test cases. However, it does not contain a software requirements specification, an architectural document nor a design document. iTrust[4], a medical application, has also been used as a traceability case. It includes a requirements specification, source code and a testing plan. Still, we did not find any design or architecture document. Antoniol et al. [ACC+02] used two case studies (LEDA and Albergate) to evaluate their traceability recovery techniques. They only used the source code and the manual pages of LEDA (a C++ framework that is freely available). No other artifacts were mentioned.

Albergate is a software system that has been developed by students based on 16 functional requirements. The Albergate case study

[4] http://agile.csc.ncsu.edu/iTrust

is not published. According to [ACC+02], Albergate includes all the documentation related to the entire software development process, but traceability links were only defined among the 16 requirements and the classes implementing them. Furthermore, the documentation of Albergate is written in Italian, which is problematic for many researchers.

5.8 Conclusion

In this paper, we presented a candidate benchmark for traceability based on an irrigation system. Among other features, our benchmark includes all the typical artifacts that are produced during the development of a software system and it provides end-to-end traceability linking among these artifacts. The benchmark data, which are publicly available, do not dependent on any specific traceability tool or technique. Therefore, researchers can use our benchmark to easily assess the effectiveness of their traceability methods. The benchmark is also convenient for the comparison of traceability methods.

The benchmark we are proposing may be a single piece of a set of benchmarks on which the traceability community is currently working. Our benchmark is mainly suitable for tasks that require a rich set of artifacts with end-to-end traceability linking. In the future, our benchmark will be completed with benchmarks featuring other characteristics such as very large data sets or traceability links evolving over time.

Chapter 6
Conclusion

6.1 Thesis Summary and Contribution

In this thesis, we address the problem of maintaining the requirements specification during software evolution. Today, keeping the requirements specification up-to-date is a manual task that is very expensive and time consuming. This is why maintainers usually apply changes to the code only, and the requirements specification rapidly becomes obsolete and useless. To reduce the effort required for maintaining requirements, we propose two approaches that support the maintainer during the update.

The first approach takes advantage of the change analysis that is done at the code level to identify the requirements that are outdated. The approach is composed of three steps. In the first step, the old and new versions of the code are compared to each

other in order to identify the changes that are likely to impact requirements. Our comparing algorithm is based on heuristics we obtained from an exploratory study that we conducted to identify the relations between changes in the source code and changes in the external behaviour of software system. For each of the changes that were identified in the previous step, a set of keywords describing the change is extracted from the code. Finally, the keywords are traced to the requirements specification in order to identify the parts that are likely to be impacted. The tracing approach is based on information retrieval methods. We have two versions of the approach. The first one was developed to address changes between any two versions of the code and generates several lists of impacted requirements, where each list relates to one changed class. The second version of the approach is meant to be used after each code commit and it generates only one list of impacted requirements per relevant commit. This version is more convenient for the maintainer, as only one list is generated, but it also requires maintainers not to include more than one conceptual change per commit. To evaluate the approach, we implemented it using a combination of tools that we developed and an existing tracing tool. We then applied the approach to two software projects that have different characteristics and evaluated the correctness of the obtained results. The results are positive as we could identify the outdated requirements with both a good precision and a good recall. The heuristics obtained from the exploratory study and the approach for identifying outdated requirements based on source code changes are the first two contributions of the thesis.

The second approach, which is based on tests, is applicable when a set of high-order tests exists and is kept up-to-date. High-order tests are usually derived from requirements, and changes in these tests usually reflect changes in requirements. On the other hand, updating tests is easier and more frequent than updating requirements, as the outdated tests fail and are easily detected. The approach we propose uses the changes in high-order tests to generate hints about requirements change. We propose different types of hints for different types of changes. In the case of feature modification or deletion, we use a set of rules for identifying the requirements that are likely to be impacted. If a new feature is added, then we propose to use execution traces of the tests to extract text describing the added feature and to identify the existing requirements that relate to the new feature. We applied the rules for identifying the outdated requirements to a small example, and found that the approach works. However, a concrete implementation of the approach for extracting text about new requirements and a more elaborated evaluation of the approach still need to be done. The approach for generating hints about requirements changes based on changes in high-order tests is the third contribution of the thesis.

An additional contribution is the development of a collection of artifacts that can be used as a benchmark for evaluating software traceability approaches and tools. The benchmark was built based on the AquaLush project, which is a software for managing an irrigation system. AquaLush originally included several artifacts describing the requirements, the design and the architecture of the software system and an implementation of the system in Java. We

complemented the existing artifacts by adding a set of unit tests and high-level tests that we developed. The different artifacts in the AquaLush project constitute the *data set* of the benchmark. We then defined a set of *tasks* to be performed on the *data set* as well as the *answer set* for these tasks. The *answer set* is composed of traceability links that we defined between the various artifacts of the system, starting from the requirements, to the design and architecture then to the code and finally to the unit tests. We also defined links between the requirements and the high-level tests. Finally, we defined a set of *measures* that can be used to evaluate the performance of traceability tools and techniques when performing the *tasks* defined in the benchmark on the benchmark's *data set*.

Code-Based and Test-Based Approaches: Discussion and Scope

The advantage of the code-based approach compared to the test-based approach is that no tests are required. In fact, all the change analysis is done at the code-level directly and is traced back to the requirements specification. For the test-based approach, we require a set of high-order tests that are complete and up-to-date. We expect the test-based approach to be more accurate than the code-based approach, because high-order tests should only be affected by changes in specified requirements. However, a comparison of the effectiveness of the two approaches on real software systems is still to be done.

The current differencing algorithm used in the code-based approach is designed for code written in object-oriented programming language. Adaptation for other types of programming languages is probably feasible, but it is out of the scope of this work. In the test-based approach, there are no preconditions concerning the programming language used.

Both approaches are meant to encourage the maintainer to update the requirements specification regularly by reducing the effort needed for the update. However, our approaches do not completely eliminate the effort needed for the update, nor do they force the maintainer to do the update. Therefore, the approaches are only useful for maintainers that are willing to keep the requirements up-to-date, but do not do it due to time and cost constrains.

Our work focuses on detecting changes in functional requirements. Covering also non-functional requirements is subject to future work. In their current status, our approaches also do not detect missing requirements in the specifications. A scenario for using tests to detect new requirements was discussed in the test-based approach. However, a concrete implementation and an evaluation of this scenario are still missing.

The existence of an initial requirements specification is a precondition for both approaches. In fact, if no specification exists at all, there is no way to identify the outdated parts or to support the maintainer in the update.

Tools

To run and evaluate our code-based approach, we developed a prototype that automates the approach. The prototype is composed of three tools: (1) a differencing tool, (2) an information retrieval-based tracing tool, and (3) an Excel tracing macro. Although we used each of these three tools separately in our experiments, they can be easily integrated in one tool if needed.

We developed the differencing tool based on a Java library to compare Java API (JDiff [Doa02]). The tool compares two versions of source code written in Java, detects the changes that are likely to impact requirements and extracts a set of keywords describing the change. As output, the tool generates a list of textual files, where each file contains the keywords relating to the relevant code changes in one class. The differencing tool is highly configurable: it allows the user to set several parameters of the approach such as (1) the elements to be used for keyword extraction, (2) the depth of the call hierarchy that is considered, and (3) the Levenshtein distance [Lev66] to be used for name comparison. We use the differencing tool in the evaluations conducted in Chapter 2 and in Chapter 3.

For the tracing, we use an existing tool that is based on information retrieval called Retro [HDS+07]. Retro takes as input two lists of textual files and returns candidate traceability links that are ranked based on the similarity between the files. We use Retro to trace the keywords generated by the differencing tool to the requirements. We use Retro in the evaluations conducted in Chapter 2 and in

Chapter 3. We also use Retro to evaluate our benchmark for traceability in Chapter 5.

The Excel tracing macro merges the links obtained from Retro for different classes into one final list using a scoring technique that we present in Chapter 3. The macro is used in the evaluation of the commit version of our code-based approach in Chapter 3.

As the current implementation of the differencing tool is based on JDiff, our prototype only works for code written in Java. However, we expect the adaptation of our tool to other object-oriented programming languages to be a pure engineering problem, as our differencing algorithm does not depend on a specific object-oriented language.

6.2 Revisiting the Research Questions

In Section 1.3 we presented five research questions that relate to the achievement of the goal of the thesis. In this section, we revisit each of the research questions and summarize the answers to them.

RQ 1: What relations exist between changes in requirements and changes in source code? In Chapter 2, we present six observations about the relation between changes in source code and changes in the functional requirements of a software system.

The observations are derived from an empirical case study that was conducted on an open source software project. We use these observations to build an approach to identify code changes that impact requirements. As a validation, we use two case studies where we run the approach built on these observations. We report on the results in Chapter 3 (Section 3.6). The positive results obtained for both case studies (accuracy always higher than 85%) are a positive indicator for the validity of the observations.

RQ 2: What is an effective way to use changes in source code to identify outdated requirements? In Chapter 2, we present an approach that we constructed to identify outdated requirements based on source code changes between two releases. We also present an extension of the same approach in Chapter 3. In the extension, the approach is adapted to be used after each code commit. Our approach analyses the changes that are applied to the code and generates a ranked list of the requirements that are likely to be impacted.

RQ 3: How successful is our code-based approach in identifying outdated requirements? The answer to this question is reported in the evaluation sections of Chapter 2 and Chapter 3. We applied our approach to two case studies and found that our approach is successful in identifying the outdated requirements with both a good recall and a good filtering of the requirements that are not outdated.

RQ 4: What information about requirements change can we obtain from changes in high-order tests? In Chapter 4, we hypothesise about the possible effects of different types of maintenance tasks on requirements and on high-order tests. Based on these hypotheses, we then explore what information about requirements change we can obtain from high-order tests. Although the hypotheses we present derive from common sense, a validation on existing software projects is still to be done.

RQ 5: How can we get automatic hints about changes in requirements based on changes in acceptance tests? In Chapter 4, we present a hint generation approach that uses changes in high-order tests and traceability links between the requirements and the high-order tests to identify the requirements that are likely to be impacted. The hint generation approach also allows identifying newly added requirements and generates information about them.

6.3 Next Steps

The results obtained so far in the evaluation of the code-based approach for identifying outdated requirements are good in terms of precision and recall. Intuitively, we expect the task of the maintainer to be much easier if the outdated requirements are identified with a good precision and a good recall. However, an evaluation of the benefits of using our approach is still missing. Such an

evaluation can be done using a controlled experiment, where we compare the effectiveness of the maintainer when updating the requirements with the help and without the help of our approach. Three groups of participants, with similar experience, are needed for the experiment. All groups will be given a list of changes that they need to implement. Participants in the first group should first update the requirements specification and then propagate the changes to the code. In the second and third groups, participants have to apply the changes to the code first and then update the requirements accordingly. Participants of the third group are provided with the tool for identifying outdated requirements, while participants of the other two groups are not. At the end, we compare the average time that each of the groups needed for applying the change, the correctness of the change in both the requirements and the code and the confidence of the participants about the changes they made.

For the test-based approach, the next step is to further elaborate it and implement it in a tool. Then it should also be applied on case studies to evaluate the correctness of the generated hints as well as their usefulness for the maintainer.

In case that the evaluation experiments proposed above would show that our approach in its current state is not very useful to the maintainer, we would have to explore the reasons for this outcome and address them. If, however, the results prove that the approach is useful, which we expect and which we hope for, then the next step would be to integrate our approach into current revision control systems so that it becomes available and easily usable for every maintainer.

Bibliography

[ACC+02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[ACCDL00] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance*, pages 40–49, 2000.

[AOH05] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, 2005.

[Bas92] Victor R. Basili. Software modeling and measurement: The goal/question/metric paradigm. Technical Report CS-TR-2956, UMIACS-TR-92-96, University of Maryland, 1992.

[BCCJG11] Eya Ben Charrada, David Caspar, Cédric Jeanneret, and Martin Glinz. Towards a benchmark for traceability. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 21–30, 2011.

[BCG10] Eya Ben Charrada and Martin Glinz. An automated hint generation approach for supporting the evolution of requirements specifications. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 58–62, 2010.

[BCKG12] Eya Ben Charrada, Anne Koziolek, and Martin Glinz. Identifying outdated requirements based on source code changes. In *Proceedings of the 20th IEEE International Requirements Engineering Conference*, pages 61–70, 2012.

[Boh02] Shawn A. Bohner. Extending software change impact analysis into COTS components. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, pages 175–182, 2002.

[BR00] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *International Conference on on Software Engineering, Future of Software Engineering Track*, pages 73–87, 2000.

[Cas11] David Caspar. A benchmark for software traceability. Bachelor's Thesis, University of Zurich, 2011.

[CDP07] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. In *Future of Software Engineering*, pages 326–341, 2007.

[CHCC03] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[CHCD+11] Jane Cleland-Huang, Adam Czauderna, Alex Dekhtyar, Olly Gotel, Jane Huffman Hayes, Ed Keenan, Greg Leach, Jonathan Maletic, Denys Poshyvanyk, Youghee Shin, Andrea Zisman, Giuliano Antoniol, Brian Berenbach, Alexander Egyed, and Patrick Maeder. Grand challenges, benchmarks, and tracelab: developing infrastructure for the software traceability research community. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 17–23, 2011.

[CHCGE10] Jane Cleland-Huang, Adam Czauderna, Marek Gibiec, and John Emenecker. A machine learning

approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 155–164, 2010.

[CHDH+06] Jane Cleland-Huang, Alex Dekhtyar, Jane Huffman Hayes, Giuliano Antoniol, Brian Berenbach, Alexander Egyed, Stephanie Ferguson, Jonathan I. Maletic, and Andrea Zisman. Grand challenges in traceability. Technical report, Tech. Rep. COET-GCT-06-01-0.9, Center of Excellence for Traceability, http://www.traceabilitycenter.org/downloads/ documents/GrandChallenges/, 2006.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[CHOT99] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–339, 1999.

[CPGS07] Walter Cazzola, Sonia Pini, Ahmed Ghoneim, and Gunter Saake. Co-evolving application code and

design models by exploiting meta-data. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1275–1279, 2007.

[DHA07] Alex Dekhtyar, Jane Huffman Hayes, and Giuliano Antoniol. Benchmarks for traceability? In *Proceedings of the International Symposium on Grand Challenges in Traceability (GCT'07/TEFSE'07)*, 2007.

[Doa02] Matthew B. Doar. JDiff – what really changed? *Java Developer's Journal*, 2002.

[Dom08] Sándor Dominich. *The Modern Algebra of Information Retrieval*. Springer Verlag, 2008.

[dSAdO05] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, pages 68–75, 2005.

[DVMW02] Theo D'Hondt, Kris Volder, Kim Mens, and Roel Wuyts. Co-evolution of object-oriented software design and implementation. In *Software Architectures and Component Technology*, volume 648 of *The Springer International Series in Engineering and Computer Science*, pages 207–224. 2002.

[EAAG08] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 53–62, 2008.

[EBJ11] Neil A. Ernst, Alexander Borgida, and Ivan Jureta. Finding incremental solutions for evolving requirements. In *Proceedings of the 19th IEEE International Requirements Engineering Conference*, pages 15–24, 2011.

[EBM11] Neil A. Ernst, Alexander Borgida, and John Mylopoulos. Requirements evolution drives software evolution. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 16–20, 2011.

[Egy03] Alexander Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.

[ERSS02] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 447–454, 2002.

[ES05] Anne Etien and Camille Salinesi. Managing requirements in a co-evolution context. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 125–134, 2005.

[Fox06] Christpher Fox. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2*. Addison Wesley, 2006.

[GBJ02] Martin Glinz, Stefan Berner, and Stefan Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[GF94] Orlena Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94 –101, 1994.

[GS05] Tony Gorschek and Mikael Svahnberg. Requirements experience in practice: Studies of six companies. In A. Aurum and C. Wohlin, editors, *Engineering and Managing Software Requirements*, pages 405–424. Springer Verlag, 2005.

[HDS05] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthekeyan Sundaram. Improving after-the-fact tracing and mapping: Supporting software quality predictions. *IEEE Software*, 22(6):30–37, 2005.

[HDS06] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[HDS+07] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram, Elizabeth Ashlee Holbrook, Sravanthi Vadlamudi, and Alain April. Requirements tracing on target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, 3(3):193–202, 2007.

[HH04] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, 2004.

[HRH05] Jameleddine Hassine, Juergen Rilling, and Jacqueline Hewitt. Change impact analysis for requirement evolution using use case maps. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 81–90, 2005.

[HWP09] Andrea Herrmann, Armin Wallnöfer, and Barbara Paech. Specifying changes only — a case study on delta requirements. In *Proceedings of the 15th International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 45–58. Springer-Verlag, 2009.

[LDLL10] Wen-Tin Lee, Whan-Yo Deng, Jonathan Lee, and Shin-Jie Lee. Change impact analysis with a goal-driven traceability-based approach. *International Journal of Intelligent Systems*, 25(8):878–908, 2010.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[Lev00] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1):15–35, 2000.

[LFdC00] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Ugo de Carlini. Recovering use case models from object-oriented code: A thread-based approach. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 108–117, 2000.

[LFOT07] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), Article 13, 2007.

[LOA00] Michelle Lee, Jefferson Offutt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to

object-oriented software. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 61–70, 2000.

[Low12] Richard Lowry. Concepts and applications of inferential statistics, 2012. Distributed online at http://vassarstats.net/textbook/.

[LR03] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[LSF03] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[LW93] Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *ECOOP' 93 — Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer Berlin Heidelberg, 1993.

[MG12] Patrick Mäder and Orlena Gotel. Towards automated traceability maintenance. *Journal of Systems and Software*, 85(10):2205–2227, 2012.

[MGP08] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Enabling automated traceability maintenance by recognizing development activities applied to models. In

Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 49–58, 2008.

[MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views. *Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.

[MM03] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.

[MSW12] Andrew Meneely, Ben Smith, and Laurie Williams. iTrust electronic health care system case study. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability,* Appendix B, pages 425–438. Springer Verlag, 2012.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 13–22, 2005.

[Mye76] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1976.

[NDCAC11] Anh Nguyen Duc, DanielaS. Cruzes, Claudia Ayala, and Reidar Conradi. Impact of stakeholder type and collaboration on issue resolution time in OSS projects. In ScottA. Hissam, Barbara Russo, ManoelG. Mendonça Neto, and Fabio Kon, editors, *Open Source Systems: Grounding Research*, volume 365 of *IFIP Advances in Information and Communication Technology*, pages 1–16. Springer Berlin Heidelberg, 2011.

[OD08] David L. Olson and Dursun Delen. *Advanced data mining techniques*. Springer, 2008.

[OGPDL10] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, pages 68–71, 2010.

[PK04] Barbara Paech and Kirstin Kohler. Task-driven requirements in object-oriented development. In Julio Cesar Sampaio Prado Leite and Jorge Horacio Doorn, editors, *Perspectives on Software Requirements*, volume 753 of *The Springer International Series in Engineering and Computer Science*, pages 45–67. Springer, 2004.

[R D10] R Development Core Team. *R: A Language and Environment for Statistical Computing*, 2010. ISBN 3-900051-07-0.

[Rei02] Steven P. Reiss. Constraining software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 162–171, 2002.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pages 74–83, 2003.

[SHD05] Senthil Karthikeyan Sundaram, Jane Huffman Hayes, and Alexander Dekhtyar. Baselines in requirements tracing. In *Proceedings of the 2005 workshop on Predictor models in software engineering*, pages 1–6, 2005.

[SHR07] Maryam Shiri, Jameleddine Hassine, and Juergen Rilling. A requirement level modification analysis support framework. In *Third International IEEE Workshop on Software Evolvability*, pages 67–74, 2007.

[SW08] Mark Sherriff and Laurie Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 268–277, 2008.

[SZ05] George Spanoudakis and Andrea Zisman. *Software Traceability: A Roadmap*. Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing, 2005.

[Tic98] Walter F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[WH06] R.J. Wieringa and J.M.G. Heerkens. The methodological soundness of requirements engineering papers: a conceptual framework and two case studies. *Requirements Engineering*, 11:295–307, 2006.

[WN94] Robert Watkins and Mark Neal. Why and how of requirements tracing. *IEEE Software*, 11(4):104–106, 1994.

[WXM+] Laurie Williams, Tao Xie, Andy Meneely, Lauren Hayward, and Jason King. iTrust medical care requirements specification. Versions of September 3rd, 2010: `agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=requirements&rev=1283530873` and of February 7th, 2011: `rev=1297120633`.

[You05] Edward Yourdon. *Just enough structured analysis*. http://yourdon.com/strucanalysis/, 2005.

[YWM+05] Yijun Yu, Yiqiao Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J.C.S. do Prado Leite. Reverse engineering goal models from legacy code. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 363–372, 2005.

[ZG03] Didar Zowghi and Vincenzo Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information & Software Technology*, 45(14):993–1009, 2003.

Appendix A
Publications

This appendix presents the list of publications on which this cumulative dissertation is built.

A.1 Journal articles

[BCKG13] Eya Ben Charrada, Anne Koziolek, and Martin Glinz. A code-based approach to support requirements maintenance during software evolution. In *Journal of Software: Evolution and Process*, 2013. Under Review

A.2 Conference Papers

[BCKG12] Eya Ben Charrada, Anne Koziolek, and Martin Glinz. Identifying outdated requirements based on source code changes. In *Proceedings of the 20th IEEE International Requirements Engineering Conference*, pages 61–70, 2012.

A.3 Workshop Papers

[BCG10] Eya Ben Charrada and Martin Glinz. An automated hint generation approach for supporting the evolution of requirements specifications. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 58–62, 2010.

[BCCJG11] Eya Ben Charrada, David Caspar, Cédric Jeanneret, and Martin Glinz. Towards a benchmark for traceability. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 21–30, 2011.

A.4 PhD Symposium

[Ben10] Eya Ben Charrada. Updating Requirements from Tests during Maintenance and Evolution. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 337–340, 2010

Curriculum Vitae

Name:	Eya Ben Charrada
Date of Birth:	August 26, 1984
Nationality:	Tunisian

2009 - 2013	Assistant and Doctoral Studies at the *University of Zurich*, Switzerland
2005 - 2008	Engineering and Master Studies at *Ecole des Mines de Saint-Etienne*, France
2003 - 2005	Studies at *Institut Préparatoire aux Etudes Scientifiques et Techniques*, Tunis, Tunisia
1996 - 2003	High school, Sousse, Tunisia